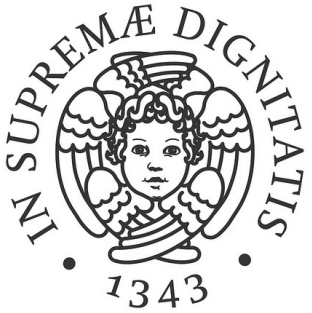

UNIVERSITÀ DI PISA

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica per l'economia e per l'azienda
(Business Informatics)

TESI DI LAUREA

PROGETTAZIONE E SVILUPPO DI UN SISTEMA PER LA GESTIONE DI LOG DI EVENTI

RELATORE
Prof. Roberto BRUNI

Candidato
Francesco SANTERAMO

ANNO ACCADEMICO 2014-15

Riassunto

I processi di business ricoprono nel mondo di oggi una parte fondamentale, guidando i servizi e le funzioni delle aziende e delle organizzazioni di tutto il mondo. Mentre la maggior parte degli attuali sistemi informativi aziendali dispone già di funzioni per l'esecuzione di tali processi e la loro registrazione tramite log di eventi, ancora molto deve essere fatto per quanto riguarda il monitoraggio e l'analisi di queste esecuzioni. In questo ambito si inserisce la disciplina del *Process Mining*, capace per mezzo di innovative tecniche di analisi e monitoraggio dei processi di entrare con successo in questa area di ricerca. Il lavoro svolto si inserisce nel contesto di questa disciplina e nasce dalla necessità di avere un sistema che permetta di creare facilmente, manipolare ed esaminare log di eventi, anche a fini didattici, utilizzando come tecnologia XES, dal 2010 lo standard XML ufficiale per la memorizzazione di log di eventi per quanto riguarda il Process Mining. L'obiettivo di questa tesi è quindi la progettazione e lo sviluppo di una applicazione basata sullo standard XES per la creazione, modifica ed analisi di log di eventi memorizzati utilizzando questo formato. Molte delle funzionalità che sono state pensate per questo nuovo sistema non sono presenti negli altri programmi disponibili per la gestione ed analisi dei log. L'obiettivo del progetto è stato quello di fornire nuove funzionalità a questa area di ricerca, complementando ed integrandosi allo stesso tempo con le altre applicazioni già presenti in questo settore.

Indice

Capitolo 1

INTRODUZIONE

1.1 Ambito del lavoro di tesi

I processi ricoprono nel mondo di oggi una parte fondamentale, guidando i servizi e le funzioni delle aziende, degli enti governativi e delle organizzazioni di tutto il mondo.

Si definisce processo aziendale un insieme di attività collegate tra loro, svolte all'interno dell'azienda, che creano valore trasformando delle risorse, che costituiscono l'input del processo, in un prodotto, che costituisce l'output del processo, quest'ultimo destinato a sua volta ad un soggetto che può essere l'azienda stessa o un soggetto esterno, ad esempio un cliente.

Sia le risorse in input che il prodotto in output possono essere beni, servizi o informazioni, oppure anche una combinazione di questi elementi.

La trasformazione dell'input del processo in output può essere svolta sia utilizzando manodopera, sia impiegando delle macchine, oppure entrambi.

Mentre la maggior parte degli attuali sistemi informativi aziendali dispone già di funzioni per l'esecuzione di tali processi, ancora molto deve essere fatto per quanto riguarda il monitoraggio e l'analisi di queste esecuzioni.

In questo ambito si inserisce la disciplina del *Process Mining*, capace per mezzo di innovative tecniche di analisi e monitoraggio dei processi di entrare con successo in questa area ancora da esplorare.

Il Process Mining è un'area di ricerca relativamente giovane che si posiziona tra la *Computational Intelligence* e il *Data Mining*, da un lato, e tra la modellazione e l'analisi dei processi, dall'altro.

Il processo di analisi parte da un log di eventi, che altro non è che una registrazione sequenziale di attività, ciascuna che identifica un passaggio ben preciso di un processo in esecuzione, raggruppate per istanze di esecuzione del processo, ovvero sequenze di attività che elencano i passaggi effettuati dall'inizio alla fine del processo durante una sua singola esecuzione.

L'insieme di queste istanze rappresenta le possibili esecuzioni del processo e costituisce il log di eventi oggetto di analisi del Process Mining.

L'obiettivo del Process Mining è quello di dedurre, monitorare e migliorare processi reali attraverso tecniche di *discovery*, estraendo un modello di processo a partire da un log di eventi senza utilizzare alcuna informazione a priori, di *conformance checking*, monitorando eventuali differenze tra il modello di processo generato mediante discovery e il log in analisi (che rappresenta le esecuzioni effettive del processo) per scoprire e capire eventuali deviazioni e misurarle, e di *enhancement*, migliorando o estendendo il modello esistente per mezzo di ulteriori informazioni riguardanti il processo, estratte sempre dal log di eventi.

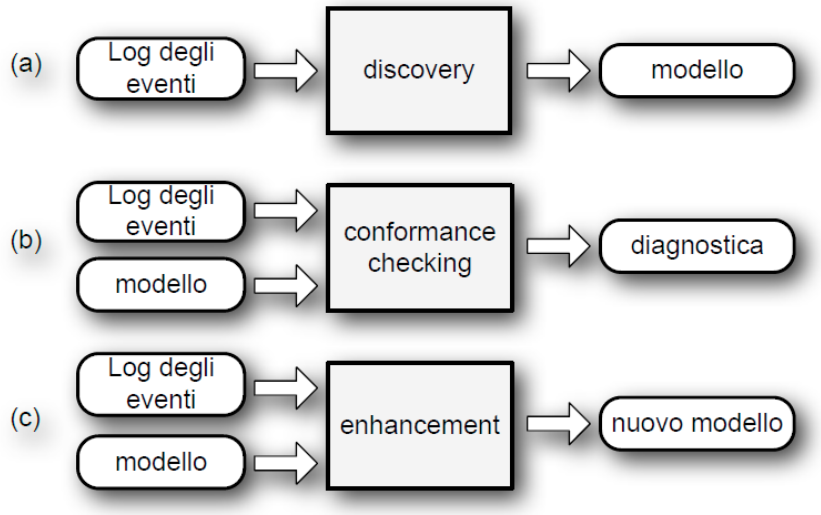

Figura 1.1: I tre tipi base di Process Mining in termini di input ed output: (a) *discovery*, (b) *conformance checking*, ed (c) *enhancement* [2]

In Figura 1.1 vengono mostrate le tre tecniche di Process Mining appena citate, in termini di input/output. Si può notare come le tecniche di discovery producano, a partire da un log di eventi, un modello che lo rappresenti. Gli standard utilizzati per la definizione del modello possono essere molteplici, ad esempio una rete di Petri, un modello BPMN[1], un modello EPC[2] o un diagramma UML delle attività. Questo modello, insieme al log di eventi, può essere testato attraverso tecniche di conformance checking, ottenendo informazioni diagnostiche che mostrano eventuali difformità tra i due input, oppure, grazie alle tecniche di enhancement, si può generare una versione migliorata ed estesa del modello stesso utilizzando il log di eventi come rappresentazione della realtà al quale il modello deve essere rapportato.

Il motivo per il quale questa disciplina suscita sempre più interesse nel-

[1] *Business Process Model and Notation.*
[2] *Event-driven Process Chain.*

l'ambito dell'analisi dei dati è dovuto da un lato alla grande quantità di dati disponibili che contengono informazioni dettagliate sulle passate esecuzioni dei processi, dall'altro alla necessità di migliorare e supportare i processi aziendali in ambienti competitivi e in rapida evoluzione.

I modelli di processi che vengono generati sono inoltre più efficaci dei semplici log per la comprensione delle dinamiche del processo. I modelli infatti sono più facili da interpretare rispetto ai singoli log, in quanto ne forniscono una rappresentazione sulla quale si possono utilizzare tecniche di analisi e simulazione, ottenendo risultati più indicativi sul processo in analisi.

Grazie alla veloce crescita tecnologica della società moderna e alla sempre più ricorrente digitalizzazione dei dati (sia per quanto concerne la loro registrazione, sia per quanto riguarda il loro scambio elettronico), le aziende hanno adesso la possibilità di registrare e successivamente analizzare gli eventi che caratterizzano il loro business. Questo diventa di fondamentale importanza nel momento in cui i dati dei log possono essere sfruttati per creare conoscenza, elemento determinante per qualsiasi azienda moderna per avere un business di successo ed essere leader del proprio settore.

I log di eventi possono avere forme e strutture differenti, in quanto ogni sistema informativo che abbia al suo interno un sistema di registrazione di log, ha da sempre sviluppato il suo standard interno per questo compito, di fatto rendendo i suoi log non analizzabili da sistemi esterni, ma solo da eventuali programmi sviluppati appositamente per quel sistema informativo o per quella struttura di log.

Questa disomogeneità complica lo sviluppo di strumenti di analisi universali, in grado di gestire qualsiasi log di eventi.

Per risolvere questo problema è stato creato un formato per la registrazione di log di eventi, standardizzato dalla IEEE[3] *Task Force on Process Mining*[4] nel 2010 ed adottato come formato di default per codificare e scambiare log di eventi, chiamato XES[5].

XES è uno standard[6] XML per la memorizzazione di log di eventi: il suo scopo è quello di definire un formato generalmente riconosciuto per permettere lo scambio di dati (riguardanti log di eventi) tra i sistemi informativi che li registrano e gli strumenti che li gestiscono e analizzano, soprattutto nel caso di sistemi eterogenei sviluppati da società diverse. Rimandiamo un maggiore approfondimento di questo standard alla Sezione 2.1, quando parleremo delle tecnologie utilizzate per lo sviluppo dell'applicazione oggetto di questo documento.

Il lavoro svolto si inserisce nel contesto di questo standard e nella necessità di avere un sistema che permetta di creare facilmente, manipolare ed esaminare log di eventi a fini didattici per i quali lo standard XES non è adatto, in quanto difficilmente interpretabile da un essere umano.

I contenuti di questa sezione riprendono i concetti espressi in [1, 2].

[3] *Institute of Electrical and Electronic Engineers.*

[4] Gruppo creato dalla IEEE per promuovere la ricerca, lo sviluppo, l'educazione e la conoscenza del Process Mining. Nel contesto di questa Task Force, un gruppo di più di 75 persone che coinvolgono più di 50 organizzazioni diverse ha creato il *Manifesto sul Process Mining*: un insieme di principi fondamentali ed importanti sfide che devono servire da guida per tutti i professionisti del settore, nell'intento di migliorare la conoscenza del Process Mining come nuovo metodo per la (ri)modellazione, il monitoraggio ed il supporto dei processi di business.

[5] *eXtensible Event Stream.*

[6] www.xes-standard.org.

1.2 Obiettivo della tesi

L'obiettivo di questa tesi è la progettazione e lo sviluppo di una applicazione per la creazione, modifica ed analisi di log di eventi.

Il sistema sviluppato è basato sullo standard XES e permette la gestione di log di eventi memorizzati utilizzando questo formato.

L'applicazione dà la possibilità di aprire, modificare e validare log di eventi reali utilizzando un editor versatile e di generarne di nuovi, definendo, attraverso un linguaggio di espressioni, possibili sequenze di eventi che identificano l'esecuzione di un processo. La generazione di nuovi log può essere fatta anche attraverso una specifica sintassi che è stata introdotta all'interno del sistema.

Questa applicazione permette inoltre di analizzare ogni log aperto e di generare una *footprint matrix* dello stesso, ovvero una matrice che mostra le dipendenze causali tra gli eventi che compongono il log e che può essere sfruttata da tecniche di discovery per creare modelli di processi [1, 3]. L'analisi può essere ulteriormente affinata attraverso l'utilizzo di vari filtri messi a disposizione, in modo da ottenere più "conoscenza" possibile dal log analizzato, scartando ad esempio tracce infrequenti o incomplete.

Sono disponibili inoltre vari formati per il salvataggio dei log su file esterni, nonché il loro export in formato CSV e l'export delle footprint matrix generate in formato HTML, PDF, PNG e LaTeX[7]. Insieme alla funzione di export, ne viene fornita una di import per la generazione di log di eventi a

[7]Linguaggio di *markup* utilizzato per la preparazione di testi basato sul programma di composizione tipografica TEX.

partire da documenti esterni in formato CSV e per il caricamento nel sistema di footprint matrix precedentemente esportate.

Molte delle funzionalità elencate sono state pensate ed implementate all'interno di questo sistema perché attualmente non sono disponibili negli altri programmi presenti online per la gestione ed analisi dei log, ad esempio nel framework ProM (un noto sistema di Process Mining), nell'intento di fornire nuove funzionalità a questa area di ricerca e di integrarsi con le altre applicazioni già presenti in questo ambito.

L'applicazione è stata sviluppata in Java e verrà rilasciata con licenza Open Source con il nome di *Event Log Manager*.

1.3 Contenuto della tesi

Di seguito viene descritta la struttura della relazione di tesi e l'organizzazione dei contenuti.

Nel Capitolo 2 vengono brevemente introdotte e discusse le tecnologie utilizzate che costituiscono le fondamenta per lo sviluppo di questa applicazione.

Nel Capitolo 3 mostreremo la struttura dell'applicazione e come questa sia stata progettata, quali pattern sono stati utilizzati e come le varie componenti che caratterizzano il sistema interagiscano tra loro.

Nel Capitolo 4 approfondiremo la descrizione dell'applicazione, presentando tutte le funzioni sviluppate e disponibili per la gestione dei documenti, con una analisi sulla loro interconnessione e robustezza.

Nel Capitolo 5 approfondiremo le funzioni dell'applicazione destinate alla generazione assistita di log, che includono un linguaggio di espressioni per definire insiemi di tracce e una sintassi compatta in formato CSV per la manipolazione di log, all'analisi dei processi e non ultimo alla gestione e trasformazione dei documenti in altri formati disponibili. In questo capitolo parleremo anche delle problematiche rilevate sullo standard XES e delle soluzioni adottate.

Nel Capitolo 6 mostreremo le funzioni per la gestione dell'interfaccia grafica dell'applicazione e quelle rese disponibili all'utente per una interazione diretta con quest'ultima.

Nel Capitolo 7 mostreremo alcuni casi d'uso riguardanti le operazioni tipiche che un utente può svolgere con questo sistema di gestione ed analisi.

Infine, nel Capitolo 8 ricapitoleremo brevemente i risultati conseguiti e descriveremo i possibili sviluppi futuri.

Capitolo 2

TECNOLOGIE UTILIZZATE

Il progetto di tesi si basa su di una serie di tecnologie che sono state fondamentali per lo sviluppo dell'intera applicazione.

In questo capitolo presenteremo per prima cosa lo standard XES, uno standard XML per la memorizzazione di log di eventi, che rappresenta uno degli elementi portanti dell'applicazione, in quanto è lo standard che definisce la struttura dei log che questa applicazione può creare, modificare ed analizzare ed è lo stesso standard utilizzato anche da altre applicazioni, note nel campo del Process Mining, come ad esempio ProM.

Per la stesura di questa sezione riguardante lo standard XES abbiamo fatto riferimento a [7].

Successivamente definiremo le tecnologie utilizzate per creare il nostro Event Log Manager: il linguaggio di programmazione Java, l'ambiente di sviluppo *Eclipse* ed infine le librerie Open Source che sono state incluse nel progetto per l'implementazione di alcune delle sue funzionalità, motivandone

la scelta.

2.1 Standard XES

I log di eventi possono avere strutture molto diverse tra loro, in base al sistema informativo che li registra e all'uso che ne viene fatto. Spesso vengono utilizzati infatti formati proprietari o ad hoc per determinati sistemi informativi o di analisi.

L'obiettivo dello standard XES[1] è quello di definire una struttura comune per la definizione e memorizzazione di log di eventi, in modo che ci possa essere una interazione tra i sistemi informativi che li registrano e i sistemi che li processano e analizzano, attraverso l'utilizzo comune di questo standard.

XES è uno standard XML ed è stato standardizzato dalla IEEE[2] *Task Force on Process Mining* nel 2010 e adottato come formato di default per codificare e scambiare log di eventi.

Si basa su quattro principi fondamentali, utilizzati durante la sua creazione come linee guida:

1. Semplicità (*Simplicity*): utilizzare la soluzione più semplice possibile per rappresentare le informazioni. Un log XES dovrebbe essere facile da decodificare (*parse*) e generare, e allo stesso tempo facilmente leggibile da un essere umano (*human-readable*). Nella progettazione di questo standard, è stato ricercato il metodo migliore per rendere

[1] *eXtensible Event Stream.*
[2] *Institute of Electrical and Electronic Engineers.*

la struttura del log più significativa possibile ed al contempo di facile implementazione.

2. Flessibilità (*Flexibility*): lo standard XES dovrebbe essere capace di catturare log di eventi a partire da qualsiasi tipo di dominio di applicazione, non importa quale sia il settore di appartenenza dei processi osservati o il sistema informativo che li registra. XES infatti si propone di essere uno standard non solo per il Process Mining e i processi aziendali (*business processes*), ma anche più in generale per tutti i tipi di log di eventi, anche quelli che non riguardano i processi aziendali (*event log data*).

3. Estensibilità (*Extensibility*): in futuro dovrà essere facile estendere lo standard. L'estensione di quest'ultimo dovrà essere la più trasparente possibile, mantenendo allo stesso tempo la compatibilità con le versioni precedenti e successive. Allo stesso modo dovrà essere possibile estendere lo standard per casi speciali, come ad esempio per particolari settori di applicazione dei log di eventi o specifiche implementazioni di sistemi informativi che lavorano con i log XES.

4. Espressività (*Expressivity*): Nello sforzo di costruire un formato generico valido per qualsiasi situazione, si dovrà sempre e comunque far sì che i log di eventi che verranno memorizzati in XES abbiano la minor perdita di informazioni possibile: ricercare quindi da un lato un formato generico valido per più situazioni possibili, dall'altro permettere ad ogni sistema informativo che memorizza log utilizzando XES di avere a disposizione tutti gli elementi possibili per registrare tutte le infor-

mazioni del log di eventi. In conseguenza di questo, tutti gli elementi XES per la registrazione delle informazioni devono essere fortemente tipizzati (*strongly typed*) e deve esistere un metodo generico per appendere ad essi una semantica facile da interpretare (*human-interpretable semantics*).

Per cercare di mantenere lo standard XES un formato generico di interscambio di informazioni, soltanto gli elementi che possono essere ritrovati in qualsiasi tipo di log sono esplicitamente definiti.

Per la registrazione di ulteriori informazioni proprie di specifici sistemi, lo standard demanda infatti tale compito all'implementazione di opportune estensioni aggiuntive, tali da definire la semantica delle informazioni da registrare, in modo da poter standardizzare gli elementi che le registreranno e permettere così di scambiare con altri sistemi anche questi ulteriori dati.

2.1.1 La terminologia utilizzata

Nelle successive sezioni faremo largo uso di alcuni termini e concetti che definiremo in questa sezione in modo da non doverne dare spiegazione in seguito.

Un processo è un insieme di casi, o istanze.

Un caso consiste in una serie di eventi, che a loro volta appartengono ad un unico specifico caso.

Gli eventi all'interno di una caso sono ordinati e possono avere attributi. Tipici esempi di attributi possono essere il nome dell'evento, il tempo di esecuzione, il costo, la risorsa che l'ha generato e così via.

Assumiamo che sia possibile registrare sequenzialmente gli eventi tali che ogni evento si riferisca ad una specifica attività, ovvero ad un preciso passaggio del processo, e che questa attività sia collegata ad un particolare caso.

Quello che otteniamo è un log di eventi, che da ora in poi chiameremo per semplicità log, ovvero un insieme di casi o istanze di processo, chiamati tracce, contenenti ciascuna a loro volta un insieme ordinato di attività, chiamate eventi, che sono state eseguite durante quell'istanza di processo.

2.1.2 Il meta-modello di XES

Verrà adesso analizzato nel dettaglio il meta-modello per lo standard XES, di cui viene mostrato in Figura 2.1 il diagramma UML 2.0 delle classi.

2.1.2.1 Struttura di base

La gerarchia di base di un documento XES segue la struttura universalmente utilizzata per la registrazione di informazioni su log di eventi.

Qui di seguito riportiamo il codice semplificato di un log che utilizzeremo come esempio durante questa parte del capitolo, andando a completarne il codice man mano che la descrizione degli elementi dello standard prosegue.

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <log xes.version="2.0" xes.features="arbitrary-depth" xmlns="http://www.xes-standard.
      org/">
3    <trace>
4       <event>
5           ...
6       </event>
7       <event>
```

```
 8            ...
 9        </event>
10    </trace>
11 </log>
```

Log

Al primo livello della gerarchia si trova il *Log*, un oggetto che conterrà tutte le informazioni sugli eventi che sono relativi ad uno specifico processo.

Più nel dettaglio, il log rappresenta la registrazione delle attività di un processo che stiamo analizzando, del quale procederemo a registrare ogni esecuzione, memorizzando per ciascuna di esse le attività che sono state eseguite, e a inserirla all'interno del log stesso, che quindi conterrà tutte le attività compiute durante le varie esecuzioni del processo, divise per istanza di esecuzione.

Il tag utilizzato per questo oggetto nella rappresentazione XML di XES è `<log>`.

Nella Tabella 2.1 sono mostrati i possibili attributi di questo tag.

Un esempio di intestazione di un documento XES in formato XML è il seguente:

```
1 <log xes.version="2.0" xes.features="nested-attributes">
```

Trace

Un log può contenere un numero arbitrario di tracce (rappresentate da un oggetto *Trace*), oppure anche essere vuoto.

Chiave	Tipo	R*	Descrizione
xes.version	xs:decimal	Si	La versione dello standard XES a cui il documento è conformato (ad es. "2.0").
xes.features	xs:token	Si	Una lista di proprietà opzionali, separate da uno spazio bianco, che il documento utilizzerà (ad es. "nested-attributes"). Se nessuna proprietà viene definita, questo attributo deve avere un valore vuoto.

* Richiesto

Tabella 2.1: Gli attributi del tag XML `<log>`

Ogni traccia rappresenta l'esecuzione di una specifica istanza, o caso, del processo che stiamo registrando. Essa racchiude tutte le attività che sono state effettuate durante una precisa esecuzione del processo e che costituiscono i passaggi di quella specifica istanza.

Il tag utilizzato per questo oggetto nella rappresentazione XML di XES è `<trace>`. Non ci sono attributi XML per questo tipo di tag.

Event

Ogni traccia può contenere un numero arbitrario di eventi (rappresentati da un oggetto *Event*), oppure anche essere vuota.

Gli eventi rappresentano singole attività che sono state osservate durante l'esecuzione del processo e che vengono registrate nel momento della loro attuazione.

Gli eventi di per sé non hanno un valore che identifica la durata dell'attività, questo perché un evento rappresenta una attività di granularità atomica

di cui si può quindi registrare solo il momento della attuazione (quindi un *timestamp* di quando l'attività è stato registrato).

Tipicamente però molte attività hanno una durata significativa e non possono essere rappresentate solo dal momento della loro attuazione, ma deve esserne registrato il tempo di durata.

In questo caso si utilizzano due eventi, uno di inizio (*start*) e uno di fine (**end**), entrambi con granularità atomica in modo da registrare l'inizio e la fine dell'attività rappresentata.

Analogamente, attività che hanno un ciclo di vita più complesso (con possibilità di essere sospese e riattivate o ripetute) avranno un evento per ciascuna fase significativa.

Il tag utilizzato per questo oggetto nella rappresentazione XML di XES è `<event>`. Non ci sono attributi XML per questo tipo di tag.

2.1.2.2 Gli attributi

Gli oggetti definiti finora (Log, Trace ed Event) definiscono la struttura del documento ma non contengono direttamente informazioni.

Tutte le informazioni riguardanti il processo in esecuzione e le attività che lo compongono sono infatti memorizzate in oggetti *Attribute*, che descrivono di fatto gli elementi che li contengono (oggetti quali Log, Trace ed Event appunto).

Tutti gli attributi hanno una chiave (*key*) di valore testuale, definita come segue:

1. La chiave dell'attributo è una chiave testuale, può contenere spazi iniziali e finali, o più spazi, ma non può contenere caratteri di tabulazione

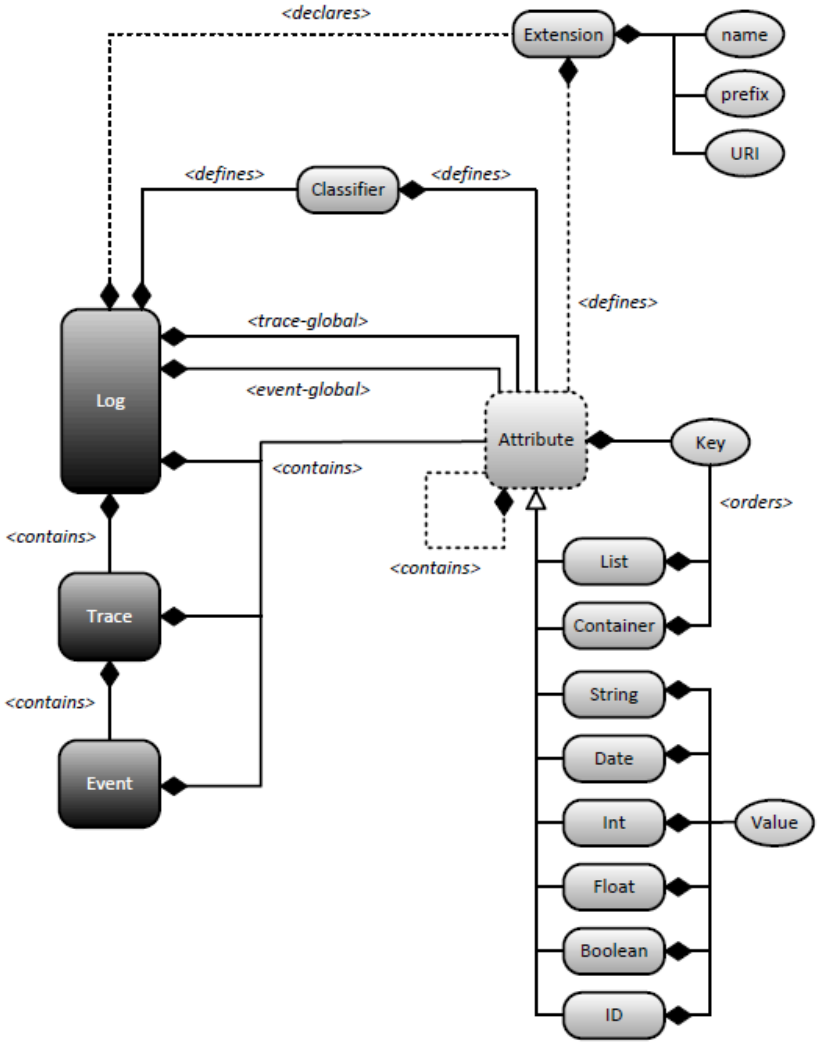

Figura 2.1: Il diagramma UML 2.0 delle classi del meta-modello per lo standard XES [7]

(*tabs*), ritorno a capo (*carriage returns*) o righe vuote (*line feeds*).

2. Lo standard XES richiede che queste chiavi siano uniche all'interno degli oggetti che le contengono (ad es. per ogni blocco `<trace>` un solo attributo con chiave "id" può esistere, ma ogni blocco traccia può

avere al suo interno un attributo con quella chiave). L'unica eccezione ammessa sono le chiavi contenute all'interno di liste (che vedremo in seguito): in questi costrutti gli attributi sono riportati in un ordine ben preciso e possono non essere unici in quanto la chiave viene ripetuta più volte all'interno della lista (ad es. una lista di oggetti Attribute con chiave "version" e valore il numero della versione, mantenuti in una lista sequenziale per registrare tutte le versioni create di un determinato software).

Log, tracce ed eventi possono contenere un qualsiasi numero di attributi.

Lo standard definisce sei tipi di attributi elementari, ciascuno di essi caratterizzato del tipo di dato che rappresenta: *String*, *Date*, *Int*, *Float*, *Boolean* e *ID*.

Insieme a questi, lo standard definisce due tipi di attributi che definiscono liste e collezioni di attributi: *List* e *Container*.

String

L'attributo *String* contiene informazioni letterali che hanno una connotazione e una lunghezza non definite.

Nella rappresentazione XML di XES, i valori di questo attributo sono memorizzati come tipo di dato `xs:string`.

Il seguente esempio mostra un possibile attributo String che definisce il nome di un elemento:

```
1 <string key="name" value="Francesco" />
```

Date

L'attributo *Date* contiene informazioni riguardanti un momento preciso nel tempo (con una precisione di millisecondi). Registra in altre parole un valore temporale che indica non solo la data, ma anche l'ora con una precisione al millisecondo e il fuso orario.

Nella rappresentazione XML di XES, i valori di questo attributo sono memorizzati come tipo di dato `xs:dateTime`.

Il seguente esempio mostra un possibile attributo Date che definisce il timestamp di un attività:

```
1 <date key="timestamp" value="2015-04-25T19:45:32.345+02:00" />
```

Int

L'attributo *Int* registra un numero intero discreto con una precisione di tipo *long* a 64bit.

Nella rappresentazione XML di XES, i valori di questo attributo sono memorizzati come tipo di dato `xs:long`.

Il seguente esempio mostra un possibile attributo Int che registra il valore di un contatore:

```
1 <int key="counter" value="236366" />
```

Float

L'attributo *Float* registra un numero continuo a virgola mobile (*floating-point*) con una precisione di tipo *double* a 64bit.

Nella rappresentazione XML di XES, i valori di questo attributo sono memorizzati come tipo di dato `xs:double`.

Il seguente esempio mostra un possibile attributo Float che registra un valore percentuale:

```
1 <float key="percentage" value="75.68" />
```

Boolean

L'attributo *Boolean* contiene un valore booleano che può essere "true" oppure "false".

Nella rappresentazione XML di XES, i valori di questo attributo sono memorizzati come tipo di dato `xs:boolean`.

Il seguente esempio mostra un possibile attributo Boolean che memorizza il completamento o meno di una operazione:

```
1 <boolean key="completed" value="true" />
```

ID

L'attributo *ID* contiene un valore che rappresenta un identificatore, di solito un UUID[3].

Nella rappresentazione XML di XES, i valori di questo attributo sono memorizzati come tipo di dato `xs:string`.

Il seguente esempio mostra un possibile attributo ID che memorizza l'identificatore univoco associato ad un determinato cliente:

```
1 <id key="customer" value="f81d4fae-7dec-11d0-a765-00a0c91e6bf6" />
```

[3] *Universally Unique Identifier*: è un identificatore standard usato nelle infrastrutture software, standardizzato dalla *Open Software Foundation*.

List

L'attributo *List* può contenere un qualsiasi numero di sotto-attributi (*child attributes*).

Questi sotto-attributi sono ordinati e le loro chiavi non sono univoche (si può ripetere la stessa chiave più volte all'interno dell'attributo List).

Il valore di questo attributo è dato dai valori degli attributi che a sua volta contiene.

Il seguente esempio mostra un possibile attributo List contenente una lista di revisioni effettuate ad un documento:

```
<list key="revisions">
    <string key="name" value="XES standard" />
    <boolean key="stable" value="true" />
    <string key="revision" value="2.0" />
    <string key="revision" value="1.4" />
    <string key="revision" value="1.3" />
    <string key="revision" value="1.2" />
    <string key="revision" value="1.1" />
    <string key="revision" value="1.0" />
</list>
```

Container

L'attributo *Container* può contenere un qualsiasi numero di sotto-attributi (*child attributes*).

La differenza con l'attributo List è che in questo caso i sotto-attributi non sono ordinati e le chiavi sono univoche.

Il valore di questo attributo è dato dai valori degli attributi che a sua volta contiene.

Il seguente esempio mostra un possibile attributo Container contenente una serie di valori per la definizione di un indirizzo:

```
<container key="location">
    <string key="street" value="Largo Bruno Pontecorvo" />
    <int key="number" value="3" />
    <string key="zip" value="56127" />
    <string key="province" value="PI" />
    <string key="city" value="Pisa" />
    <string key="country" value="Italy" />
</container>
```

2.1.2.3 Attributi annidati

Lo standard XES permette, per fornire la massima flessibilità per quanto riguarda la memorizzazione dei dati, l'utilizzo di attributi annidati (*nested attributes*), ovvero ciascun attributo può a sua volta avere sotto-attributi.

Abbiamo visto come questa specifica sia necessaria per poter avere costrutti come gli attributi List e Container, ma la possibilità non è limitata solo a questi due.

Ad esempio, nel caso di un attributo String, la possibilità di inserire degli attributi annidati viene implementata nel modo seguente:

```
<string key="city" value="Piisa">
    <boolean key="spell checked" value="false" />
</string>
```

In questo caso abbiamo unito all'informazione registrata dall'attributo String (il nome di una città, quella di Pisa, ma scritto in maniera non corretta), un valore booleano che ci dice se il nome è stato o meno scritto correttamente.

In maniera del tutto simile si possono creare attributi annidati per i restanti cinque tipi di attributo di base.

Se un documento utilizza attributi annidati, questo va specificato nell'attributo `xes.features` del tag `<log>`, utilizzando il valore "nested-attributes", in modo da comunicare questa informazione al sistema che analizzerà il log.

La funzionalità degli attributi annidati è opzionale per i sistemi che elaborano documenti in formato XES: non è cioè necessario che questi sistemi siano in grado di creare e gestire attributi annidati per essere conformi allo standard XES.

Se un sistema non supporta gli attributi annidati, basterà che sia in grado di leggere il documento (e quindi gli attributi annidati presenti in esso) e riconoscerne la corretta forma (quindi ritenendo eventualmente corretti anche gli attributi annidati); successivamente potrà in maniera trasparente ignorare e scartare questi attributi, eventualmente informando l'utente che alcune informazioni non possono essere elaborate.

2.1.2.4 Attributi globali

Ciascun log può contenere nella sua parte iniziale, prima dell'inizio delle tracce, due liste speciali di attributi, una per le tracce e una per gli eventi.

Queste liste contengono una serie di attributi che vengono definiti globali (*global attributes*): sono attributi che, elencati in questo modo, dovranno essere definiti per ciascun elemento presente nel documento del rispettivo livello (per le tracce o per gli eventi).

Questo significa che, definendo un attributo come globale all'inizio del documento, ad esempio per gli eventi, questo dovrà essere presente in ogni

oggetto Event di ogni traccia. Lo stesso discorso vale per le tracce, dove un attributo globale dovrà essere presente in ogni oggetto Trace del log.

Gli attributi globali sono definiti dal tag XML `<global>`, sotto-elemento del tag `<log>` del documento.

Un esempio di attributo globale per gli eventi è il seguente:

```
1 <global scope="event">
2     <string key="name" value="" />
3 </global>
```

Questo blocco dichiara che ogni evento del log dovrà avere un attributo String con chiave il valore "name".

Nella definizione dell'attributo, il campo `value` indica il valore di default dell'attributo nel caso un evento venga creato senza questo abbia una definizione per questo valore. Può essere utilizzato nel caso in cui questo attributo debba sempre avere un valore significativo, per un qualche tipo di analisi che vogliamo effettuare, in modo che in ogni caso questo valore esista sempre. Nel caso contrario, può essere lasciato vuoto.

Gli attributi globali sono una funzionalità necessaria per il rispetto dello standard XES e le applicazioni che gestiscono log in formato XES devono saper gestire la presenza di questi attributi durante la fase di validazione del documento nel rispetto dello standard.

Qui di seguito riportiamo il nostro esempio iniziale, ulteriormente completato adesso con gli elementi visti finora.

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <log xes.version="2.0" xes.features="arbitrary-depth" xmlns="http://www.xes-standard.
      org/">
3     ...
4     <global scope="trace">
```

```
5         <string key="concept:name" value="" />
6     </global>
7     <global scope="event">
8         <string key="concept:name" value="" />
9         <date key="time:timestamp" value="1970-01-01T00:00:00.000+00:00" />
10        <string key="system" value="" />
11    </global>
12    ...
13    <float key="log attribute" value="2335.23" />
14    <trace>
15        <string key="concept:name" value="Trace number one" />
16        <event>
17            <string key="concept:name" value="Register client" />
18            <string key="system" value="alpha" />
19            <date key="time:timestamp" value="2009-11-25T14:12:45.000+02:00" />
20            <int key="attempt" value="23">
21                <boolean key="tried hard" value="false" />
22            </int>
23        </event>
24        <event>
25            <string key="concept:name" value="Mail rejection" />
26            <string key="system" value="beta" />
27            <date key="time:timestamp" value="2009-11-28T11:18:45.000+02:00" />
28        </event>
29    </trace>
30 </log>
```

2.1.2.5 Classificatori di eventi

In XES non esistono attributi con un significato ben definito a priori.

Ciascun attributo visto finora funge da contenitore di determinate informazioni, ma non chiarisce in alcun modo il significato di queste ultime, limitandosi a rappresentarle all'interno del log.

Per dare un significato alle informazioni contenute all'interno degli attributi degli eventi e soprattutto per poter confrontare questi ultimi in base proprio ai loro attributi e poter controllare se questi abbiano gli stessi valori, vengono introdotti i classificatori di eventi (*event classifiers*).

Lo standard XES introduce il concetto di classificatore di eventi per rendere la classificazione di questi elementi configurabile e flessibile: un classificatore associa a ciascun evento una identità, definendo un insieme di attributi che costituisce appunto il modo con il quale definire questa identità e che permette poi agli eventi di essere confrontati tra loro.

Nella forma più semplice, un classificatore è definito da un solo attributo, e il valore di quell'attributo definisce l'identità della classe di un evento. Per ogni valore di questo attributo all'interno del log, viene definita una classe e gli eventi sono raggruppati in queste classi in base al valore dell'attributo considerato.

I classificatori sono definiti per l'intero log e possono essere in numero arbitrario per ciascun documento.

Ciascuno di essi è definito dal rispettivo tag XML `<classifier>`, sottoelemento del tag `<log>` del documento.

Un esempio di classificatore di eventi è il seguente:

```
1 <classifier name="Activity classifier" keys="name status" />
```

In questo esempio viene definito un classificatore il cui identificatore è "Activity classifier" sull'insieme di attributi "name" e "status". Qualsiasi coppia di eventi che avrà gli stessi valori per entrambi gli attributi, sarà considerata uguale da quel classificatore.

Come si può vedere dall'esempio, l'attributo `keys` del tag `<classifier>` utilizza gli spazi per separare i nomi degli attributi (ovvero i loro identificatori definiti all'interno del loro attributo `name`), ma l'identificatore di un attributo può avere a sua volta spazi al suo interno (ad es. `name="attribute one"`). Per decidere quindi quali spazi separino le chiavi degli attributi elencati e quali invece siano contenuti in queste ultime, XES utilizza un approccio misto. Per prima cosa, viene fornita la possibilità di raggruppare le chiavi attraverso singoli apici (ad es. `keys="'attribute one' name"`). Se le chiavi così definite non corrispondono ad attributi globali per eventi, partendo dalla prima si procede unendo questa alla successiva (avremo quindi `keys="attribute one name"`), ricercando questa nuova combinazione come chiave di attributo. Se questa viene trovata, avremo che il classificatore sarà definito sopra questa nuova combinazione di chiavi, altrimenti, se né la prima né le successive combinazioni corrispondono ad attributi globali per eventi, il classificatore sarà definito sulle singole chiavi definite inizialmente, anche se queste non corrispondono appunto a nomi di attributi globali per eventi.

E' importante notare come l'insieme di attributi utilizzati per definire un classificatore dovrebbe essere un sottoinsieme degli attributi globali per gli eventi di quel determinato log. In altre parole un classificatore di eventi dovrebbe essere definito solamente sull'insieme degli attributi globali per gli eventi.

Questa specifica, sebbene non obbligatoria, risulta fondamentale ai fini dell'analisi del log, in quanto assicura che ciascun evento possa essere classificato secondo l'insieme di attributi utilizzati per definire il classificatore, poiché questi attributi sono globali per gli eventi e quindi ciascun evento do-

vrà necessariamente definirli al suo interno per rispettare lo standard XES e far risultare di conseguenza il documento come validato.

I classificatori di eventi sono una funzionalità obbligatoria dello standard XES, in quanto permettono l'analisi del log secondo specifiche classificazioni dei suoi eventi, altrimenti non paragonabili tra loro.

2.1.2.6 Le estensioni

Lo standard XES non definisce uno specifico set di attributi per il log, le tracce e gli eventi.

In conseguenza di ciò, la semantica degli attributi che questi oggetti contengono risulta ambigua, ostacolando la corretta interpretazione dei dati.

Questa ambiguità viene risolta da XES grazie al concetto di estensioni (*extensions*). Una estensione infatti definisce un insieme di attributi per ogni livello della gerarchia di XES che abbiamo visto finora (log, tracce, eventi e meta-informazioni per gli attributi annidati), in modo da fornire una struttura per interpretare questi attributi (e di conseguenza gli oggetti ai quali appartengono).

Le estensioni quindi sono per prima cosa uno strumento per associare una semantica ad un insieme di attributi di un elemento, in modo da regolarne le chiavi ed i possibili valori per una interpretazione più immediata.

Possono essere utilizzate per molteplici ragioni, una prima importante è quella di introdurre e definire un insieme di attributi comunemente riconosciuti che sono vitali per una specifica prospettiva o dimensione dell'analisi dei log di eventi. XES a tale scopo fornisce una serie di estensioni standard che definiscono insiemi specifici di attributi per vari tipi di analisi.

Grazie a queste estensioni infatti (e alle altre che possono essere definite separatamente da un sistema per casi specifici di analisi) si vengono a creare delle semantiche ben precise per gli attributi, in modo che i vari sistemi di analisi dei log di eventi, che utilizzano XES come formato per i log, possano facilmente riconoscere questi attributi e saperne interpretare ed utilizzare al meglio il valore. Vedremo nelle sezioni successive le estensioni standard che XES fornisce.

Un' altro motivo per utilizzare le estensioni è quello di definire ulteriori attributi per uno specifico settore di applicazione, attributi che di solito sono generalmente riconosciuti in questo determinato ambito ed importanti per l'analisi di specifici dati contenuti nei log memorizzati.

Nella serializzazione XML di XES, le estensioni sono dichiarate attraverso il corrispettivo tag `<extension>`, sotto-elemento del tag `<log>`.

Un esempio di dichiarazione di una estensione all'interno di un log è la seguente:

```
1 <extension name="Concept" prefix="concept" uri="http://www.xes-standard.org/concept.
      xesext" />
```

Questa riga specifica che il log utilizza attributi contenuti nell'estensione *Concept*.

Il campo `uri` della dichiarazione contiene un URI[4] univoco che rimanda alla definizione dell'estensione in formato XESEXT.

Ogni estensione definisce un personale prefisso (*prefix*) che identifica gli attributi in essa contenuti:

```
1 <string key="concept:name" value="Initialization" />
```

[4] *Uniform Resource Identifier*

In questo esempio possiamo vedere come la chiave dell'attributo sia composta dal prefisso `concept` e separata dai due punti dal nome dell'attributo definito dall'estensione.

In questo modo gli attributi fanno riferimento diretto alle loro rispettive estensioni che li definiscono e si elimina ogni possibile ambiguità riguardo a quale attributo di quale estensione ci si stia riferendo.

2.1.3 Serializzazione XML di XES

Dopo aver concluso l'esposizione dei vari componenti del meta-modello di un documento XES ed aver nominato per ciascuno di essi il rispettivo tag nella serializzazione XML di XES, viene adesso presentato un esempio di come appare tale documento nella sua forma completa.

Il *Syntax Diagram* in Figura 2.2 mostra la corretta composizione di un documento XES.

Il diagramma presentato illustra tutte le possibili combinazioni di elementi che possono far parte di un documento XES.

Qui di seguito mostriamo dunque il nostro esempio, ora completo di tutti gli elementi dello standard e conforme alla sintassi in Figura 2.2.

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <log xes.version="2.0" xes.features="arbitrary-depth" xmlns="http://www.xes-standard.
      org/">
3     <extension name="Concept" prefix="concept" uri="http://www.xes-standard.org/concept
          .xesext" />
4     <extension name="Time" prefix="time" uri="http://www.xes-standard.org/time.xesext"
          />
5     <global scope="trace">
6         <string key="concept:name" value="" />
7     </global>
```

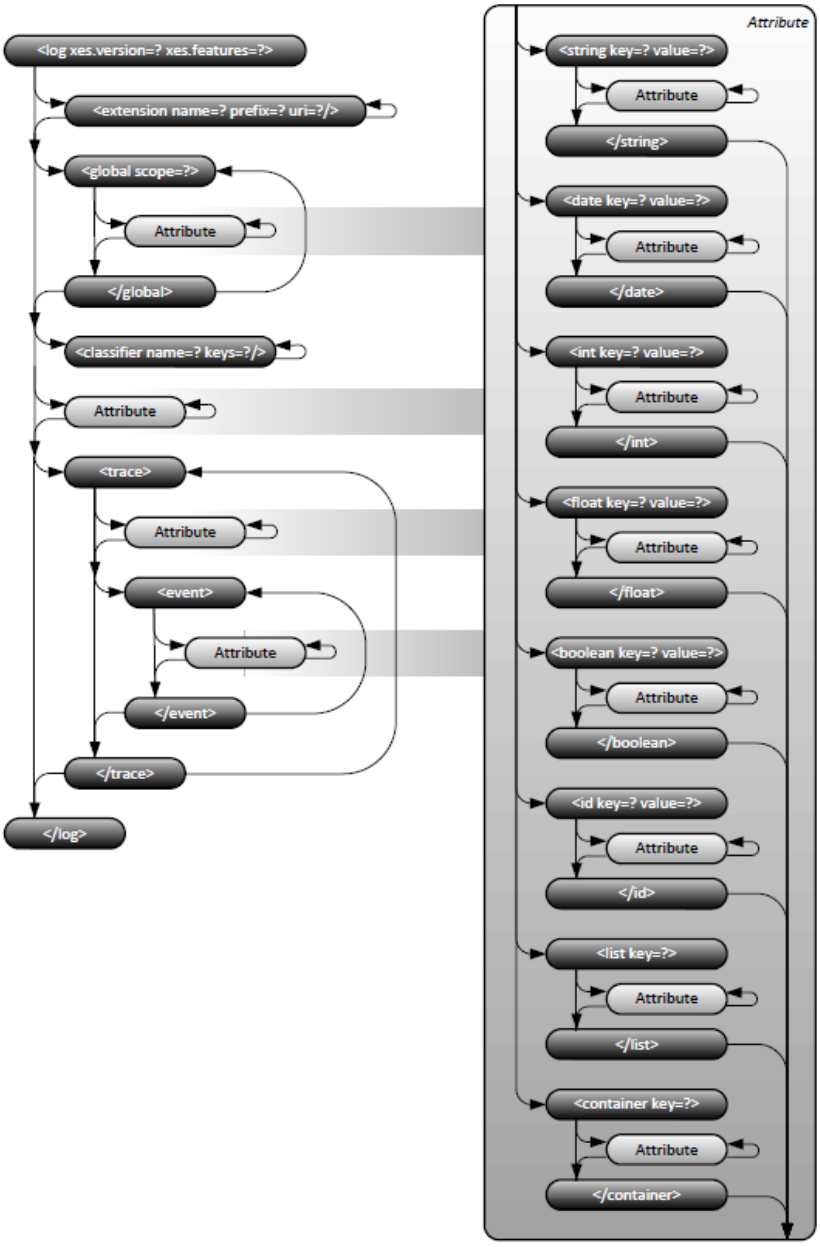

Figura 2.2: Il diagramma di flusso per lo standard XES [7]

```
8     <global scope="event">
9         <string key="concept:name" value="" />
10        <date key="time:timestamp" value="1970-01-01T00:00:00.000+00:00" />
```

```
11        <string key="system" value="" />
12    </global>
13    <classifier name="Activity" keys="concept:name" />
14    <classifier name="Another" keys="concept:name system" />
15    <float key="log attribute" value="2335.23" />
16    <trace>
17        <string key="concept:name" value="Trace number one" />
18        <event>
19            <string key="concept:name" value="Register client" />
20            <string key="system" value="alpha" />
21            <date key="time:timestamp" value="2009-11-25T14:12:45.000+02:00" />
22            <int key="attempt" value="23">
23                <boolean key="tried hard" value="false" />
24            </int>
25        </event>
26        <event>
27            <string key="concept:name" value="Mail rejection" />
28            <string key="system" value="beta" />
29            <date key="time:timestamp" value="2009-11-28T11:18:45.000+02:00" />
30        </event>
31    </trace>
32 </log>
```

2.1.4 Estensioni standard

Il meta-modello di XES riconosce e tratta tutte le estensioni come uguali, indipendentemente se facenti parte di quelle standard oppure di quelle esterne. Questo permette di estendere il formato di XES e di adattare questo standard a qualsiasi tipo di scopo e dominio di applicazione.

Vi sono comunque alcuni requisiti ricorrenti per la memorizzazione delle informazioni nei log di eventi, che necessitano una semantica universale e ben definita per essere riconosciuti da tutti i sistemi.

Per questo motivo, un insieme di sette estensioni sono state standardizzate. Utilizzare queste estensioni standard nei log di eventi permette di formalizzare il documento seguendo una semantica ben precisa e riconosciuta, permettendo un livello di conoscenza maggiore dei contenuti del log da parte dei sistemi di analisi e quindi una analisi più accurata ed approfondita dello stesso.

Verranno ora presentate le estensioni standard del formato XES.

2.1.4.1 Concept Extension

L'estensione *Concept* definisce, per tutti i livelli della gerarchia di XES, un attributo che memorizza il nome di riferimento di quel dato elemento della gerarchia, quello cioè che lo identifica.

- Prefisso: `concept`
- URI: `http://www.xes-standard.org/concept.xesext`

Nella Tabella 2.2 sono mostrati gli attributi definiti da questa estensione, suddivisi per i vari livelli della gerarichia di XES.

2.1.4.2 Lifecycle Extension

L'estensione *Lifecycle* specifica, per gli eventi, la transizione che essi rappresentano in un modello transazionale del ciclo di vita composto delle attività.

Il modello transazionale può essere di tipo generico, ciononostante questa estensione definisce anche un modello transazionale standard per le attività, mostrato nella Figura 2.3 attraverso un automa a stati.

Livello	Chiave	Tipo	Descrizione
Log, Trace, Event	name	string	Memorizza un nome identificativo per ciascun elemento della gerarchia. Per i log, il nome dell'attributo può registrare il nome del processo che è stato eseguito. Per le tracce, questo attributo memorizza di solito l'identificatore (ID) del caso registrato. Per gli eventi, l'attributo rappresenta il nome dell'evento, ovvero il nome dell'attività eseguita rappresentata da questo evento all'interno del log.
Event	instance	string	Definito per gli eventi, questo attributo rappresenta un identificatore dell'istanza dell'attività la cui esecuzione ha generato questo evento.

Tabella 2.2: Gli attributi definiti dall'estensione *Concept*

Questa estensione può essere utilizzata in qualsiasi situazione dove gli eventi rappresentano transizioni di un ciclo di vita di un insieme di attività.

- Prefisso: `lifecycle`
- URI: `http://www.xes-standard.org/lifecycle.xesext`

Nella Tabella 2.3 sono mostrati gli attributi definiti da questa estensione, suddivisi per i vari livelli della gerarichia di XES.

Nella Tabella 2.4 sono invece riportati i possibili valori dell'attributo `transition`, quando il modello transazionale del ciclo di vita selezionato è quello standard per questa estensione.

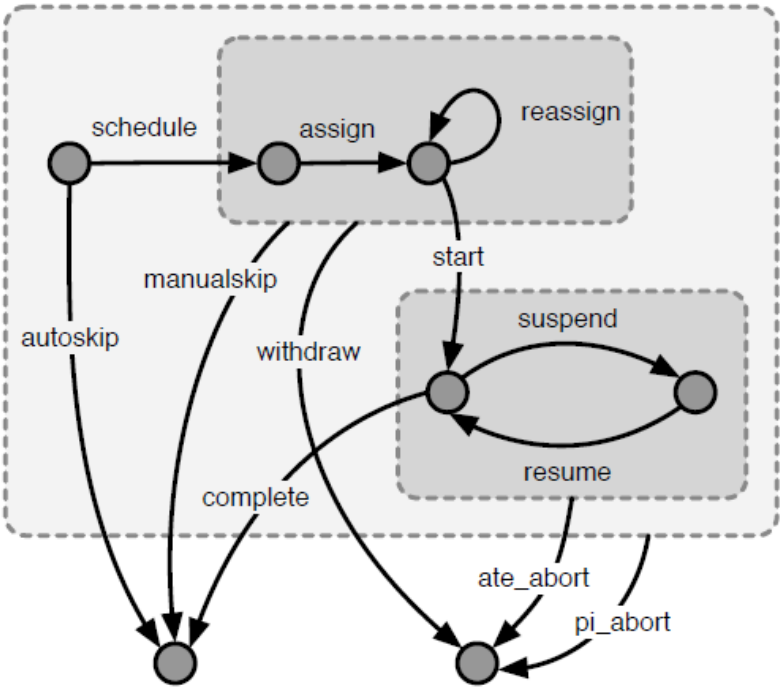

Figura 2.3: L'automa a stati per il modello transazionale standard dell'estensione *Lifecycle* [7]

2.1.4.3 Organizational Extension

L'estensione *Organizational* è utile per domini di applicazione dove gli eventi sono generati da risorse umane, che sono in qualche modo parte di una struttura organizzativa.

Questa estensione specifica tre attributi per gli eventi, i quali identificano la risorsa che ha causato l'evento e la sua posizione nella struttura organizzativa di cui fa parte.

- Prefisso: `org`
- URI: `http://www.xes-standard.org/org.xesext`

Livello	Chiave	Tipo	Descrizione
Log	model	string	Questo attributo si riferisce al modello transazionale del ciclo di vita utilizzato per tutti gli eventi del log. Se questo attributo ha come valore "standard", verrà utilizzato il modello transazionale standard del ciclo di vita definito da questa estensione.
Event	transition	string	Definito per gli eventi, questo attributo specifica la transizione del ciclo di vita rappresentata da ciascun evento. Se il modello transazionale utilizzato è quello standard definito da questa estensione, il valore di questo attributo corrisponderà ad uno di quelli riportati nella Tabella 2.4.

Tabella 2.3: Gli attributi definiti dall'estensione *Lifecycle*

Nella Tabella 2.5 sono mostrati gli attributi definiti da questa estensione, suddivisi per i vari livelli della gerarichia di XES.

2.1.4.4 Time Extension

In quasi tutte le applicazioni, l'esatta data ed orario nel quale un evento si verifica può essere registrata con precisione.

Lo scopo dell'estensione *Time* è proprio questo, registrare questa informazione, ovvero un timestamp di quando un certo evento è accaduto.

Memorizzare questo dato per gli eventi è molto importante, in quanto esso costituisce un'informazione di enorme valore per tante tecniche di analisi dei log di eventi.

- Prefisso: `time`

Valore	Descrizione
schedule	L'attività è pronta per l'esecuzione.
assign	L'attività è assegnata ad una risorsa per l'esecuzione.
withdraw	L'assegnazione è stata revocata.
reassign	Nuova assegnazione a seguito di una revoca.
start	Inizio dell'esecuzione dell'attività.
suspend	L'esecuzione è stata sospesa.
resume	L'esecuzione è stata ripresa.
pi_abort	L'intera esecuzione del processo è cancellata per questo caso.
ate_abort	L'esecuzione dell'attività è stata cancellata.
complete	L'esecuzione dell'attività è stata completata.
autoskip	Il sistema ha saltato l'esecuzione dell'attività.
manualskip	L'esecuzione dell'attività è stata saltata di proposito (manualmente).
unknown	Valore da utilizzare per tutti gli altri tipi di transizione non considerati dalle precedenti categorie.

Tabella 2.4: I possibili valori dell'attributo `transition` durante l'utilizzo del modello transazionale standard da parte dell'estensione *Lifecycle*

- URI: `http://www.xes-standard.org/time.xesext`

Nella Tabella 2.6 sono mostrati gli attributi definiti da questa estensione, suddivisi per i vari livelli della gerarichia di XES.

2.1.4.5 Semantic Extension

In base alla vista di analisi che decidiamo di avere su di un processo, gli elementi della gerarchia potrebbero corrispondere a concetti diversi.

Livello	Chiave	Tipo	Descrizione
Event	resource	string	Il nome, o identificatore, della risorsa che ha innescato questo evento.
Event	role	string	Il ruolo della risorsa che ha innescato questo evento, all'interno della struttura organizzativa di cui fa parte.
Event	group	string	Il gruppo all'interno della struttura organizzativa di cui la risorsa che ha innescato l'evento è membro.

Tabella 2.5: Gli attributi definiti dall'estensione *Organizational*

Livello	Chiave	Tipo	Descrizione
Event	timestamp	date	La data e l'orario nel quale l'evento si è verificato.

Tabella 2.6: Gli attributi definiti dall'estensione *Time*

Per fare un esempio, il nome di un evento, specificato dall'estensione Concept, potrebbe riferirsi all'attività la cui esecuzione ha generato tale evento. Questa attività potrebbe però trovarsi in un livello basso del meta-modello del processo ed essere quindi parte di un livello più alto di aggregazione.

Questo potrebbe avvenire non solo per gli eventi, ma anche per gli altri elementi della gerarchia di XES, che potrebbero fare riferimento a più concetti allo stesso tempo.

Per esprimere il fatto che un elemento della gerarchia possa avere più significati nel meta-modello del processo analizzato, si utilizza l'estensione *Semantic*.

Questa estensione suppone che esista una onotologia[5] per il meta-modello del processo, dove ogni concetto possa essere identificato da un URI univoco. In questo modo l'estensione Semantic può definire un attributo, `modelReference`, che permette la memorizzazione di un insieme di riferimenti ai vari modelli, sotto forma di URI, per ciascun elemento della gerarchia di XES.

- Prefisso: `semantic`
- URI: `http://www.xes-standard.org/semantic.xesext`

Livello	**Chiave**	**Tipo**	**Descrizione**
Log, Trace, Event, meta	modelReference	string	Riferimenti ai concetti del modello in una ontologia. I riferimenti sono memorizzati in una singola stringa di testo, registrando una sequenza di URI, separati da virgola, che identificano i concetti della ontologia.

Tabella 2.7: Gli attributi definiti dall'estensione *Semantic*

Nella Tabella 2.7 sono mostrati gli attributi definiti da questa estensione, suddivisi per i vari livelli della gerarichia di XES.

2.1.4.6 ID Extension

L'estensione *ID* fornisce identificatori univoci (nel formato UUID) per gli elementi della gerarchia di XES.

[5]In informatica, un'ontologia è una rappresentazione formale, condivisa ed esplicita di una concettualizzazione di un dominio di interesse. Più nel dettaglio, si tratta di una teoria assiomatica del primo ordine esprimibile in una logica descrittiva. Fonte Wikipedia.

- Prefisso: `identity`
- URI: `http://www.xes-standard.org/identity.xesext`

Livello	Chiave	Tipo	Descrizione
Log, Trace, Event, meta	id	id	Identificatore univoco per un elemento (UUID).

Tabella 2.8: Gli attributi definiti dall'estensione *ID*

Nella Tabella 2.8 sono mostrati gli attributi definiti da questa estensione, suddivisi per i vari livelli della gerarichia di XES.

2.1.4.7 Cost Extension

L'estensione *Cost* definisce un elemento annidato per memorizzare informazioni riguardanti i costi associati alle attività all'interno di un log.

L'obiettivo di questa estensione è cioè quello di definire una semantica per la registrazione dei costi che può essere utilizzata per gli eventi del log.

Riprendiamo il nostro esempio, aggiungendo a questo gli elementi dell'estensione Cost, in modo da rendere la descrizione di questa estensione più semplice.

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <log xes.version="2.0" xes.features="arbitrary-depth" xmlns="http://www.xes-standard.
      org/">
3    <extension name="Concept" prefix="concept" uri="http://www.xes-standard.org/concept
        .xesext" />
4    <extension name="Time" prefix="time" uri="http://www.xes-standard.org/time.xesext"
        />
5    <extension name="Cost" prefix="cost" uri="http://www.xes-standard.org/cost.xesext"/
        >
```

```
6     <global scope="trace">
7         <string key="concept:name" value="" />
8     </global>
9     <global scope="event">
10        <string key="concept:name" value="" />
11        <date key="time:timestamp" value="1970-01-01T00:00:00.000+00:00" />
12        <string key="system" value="" />
13    </global>
14    <classifier name="Activity" keys="concept:name" />
15    <classifier name="Another" keys="concept:name system" />
16    <float key="log attribute" value="2335.23" />
17    <trace>
18        <string key="concept:name" value="Trace number one" />
19        <string key="cost:currency" value="AUD" />
20        <float key="cost:total" value="20.00" />
21        <string key="xyz123" value="">
22            <float key="cost:amount" value="20.00" />
23            <string key="cost:driver" value="xyz123" />
24            <string key="cost:type" value="Fixed Overhead" />
25        </string>
26        <event>
27            <string key="concept:name" value="Register client" />
28            <string key="system" value="alpha" />
29            <date key="time:timestamp" value="2009-11-25T14:12:45.000+02:00" />
30            <int key="attempt" value="23">
31                <boolean key="tried hard" value="false" />
32            </int>
33            <string key="cost:currency" value="AUD" />
34            <float key="cost:total" value="123.50" />
35            <string key="d2f4ee27" value="">
36                <float key="cost:amount" value="21.40" />
37                <string key="cost:driver" value="d2f4ee27" />
38                <string key="cost:type" value="Labour" />
39            </string>
40            <string key="abc124" value="">
41                <float key="cost:amount" value="102.10" />
42                <string key="cost:driver" value="abc124" />
```

```
43              <string key="cost:type" value="Variable Overhead" />
44          </string>
45      </event>
46      <event>
47          <string key="concept:name" value="Mail rejection" />
48          <string key="system" value="beta" />
49          <date key="time:timestamp" value="2009-11-28T11:18:45.000+02:00" />
50          ...
51      </event>
52    </trace>
53 </log>
```

L'estensione definisce innanzitutto tre attributi annidati, ovvero attributi che dovranno dovranno essere contenuti da un altro attributo:

1. `amount`, che conterrà l'importo che definisce questo attributo di costo.
2. `driver`, che rappresenterà il fattore di costo (*cost driver*) responsabile per il sostenimento di tale costo.
3. `type` che identificherà il tipo di costo.

Come si vede nell'esempio, questi tre attributi sono utilizzati sia a livello di traccia che a livello di eventi, all'interno di un attributo principale la cui importanza è solo quella di fare da insieme per gli altri tre.

Questo attributo principale infatti ha l'unico scopo di separare in gruppi questi tre attributi, in modo da poter avere più attributi di costo per evento nel caso servisse, dato che lo standard XES non permette di avere attributi multipli con la stessa chiave all'interno dello stesso elemento.

Questi attributi di raggruppamento sono anche chiamati attributi di costo e non fanno direttamente parte di questa estensione, poiché le loro chiavi

necessitano di solito di essere differenti e possono assumere qualsiasi valore per essere distinte tra loro (e quindi non rispettano uno standard preciso come quello che una estensione si propone di definire).

Poiché possono esistere più attributi di costo associati allo stesso evento o traccia, l'estensione definisce due attributi principali a livello sia di evento che di traccia:

1. `total`, che conterrà rispettivamente la somma complessiva dei vari costi dell'evento e la somma complessiva dei totali dei singoli eventi appartenenti alla traccia.

2. `currency`, che registra la valuta nella quale i costi sono espressi.

Le informazioni sui costi possono essere quindi registrate sia a livello di tracce (ad es. per registrare che un dato caso ha generato un costo per il suo avvio), sia a livello di eventi (ad es. per eventi completati o cancellati per i quali ci interessa registrare il costo sostenuto durante l'esecuzione dell'attività che li ha generati).

Nella Tabella 2.9 sono mostrati gli attributi definiti da questa estensione e appena descritti, suddivisi per i vari livelli della gerarchia di XES.

- Prefisso: `cost`

- URI: `http://www.xes-standard.org/cost.xesext`

2.2 Sviluppo dell'applicazione

Dopo aver presentato lo standard XES, elemento fondamentale sulla quale si basa la gestione e analisi che l'applicazione sviluppata implementa, passiamo

Livello	Chiave	Tipo	Descrizione
Trace, Event	total	float	Il costo totale incorso per una traccia o un evento. Il valore rappresenta la somma di tutti gli importi registrati nell'attributo `amount` dei sotto-elementi di questo livello.
Trace, Event	currency	string	La valuta di tutti i costi di questo elemento, espressa in qualsiasi formato valido per le valute.
meta	amount	float	Il valore dell'importo per il fattore di costo dichiarato.
meta	driver	string	L'identificatore del fattore di costo utilizzato per calcolare questo elemento di costo.
meta	type	string	Il tipo di costo (ad es. *Fixed*, *Overhead*, *Materials*).

Tabella 2.9: Gli attributi definiti dall'estensione *Cost*

ora a descrivere quali sono stati gli strumenti utilizzati per lo sviluppo di quest'ultima.

2.2.1 Il linguaggio di programmazione

La prima tecnologia da definire per lo sviluppo dell'applicazione è stata quella del linguaggio di programmazione da utilizzare.

La scelta è ricaduta sul linguaggio di programmazione Java: è un linguaggio orientato agli oggetti ormai famoso in tutto il mondo e molto utilizzato anche in ambito universitario. Questa scelta ha permesso una organizzazione ottimale del codice ed implementazione dell'applicazione e permetterà una

facile manutenzione ed estensione della stessa da parte di una vasta comunità di sviluppatori e studenti universitari.

Java inoltre, come è noto in ambito informatico, è un linguaggio che permette, dopo una fase iniziale di compilazione del programma scritto per ottenere come risultato il cosiddetto *bytecode*[6], di eseguire i programmi con esso sviluppati in maniera indipendente dall'hardware sottostante: è sufficiente avere installato nel proprio sistema una piattaforma Java (di cui ne è l'implementazione il *Java Runtime Environment*, o JRE[7]), che al giorno d'oggi è presente nella quasi totalità dei sistemi operativi.

Questa piattaforma prende il bytecode generato dalla compilazione del codice Java e lo "interpreta", utilizzando la macchina virtuale inclusa nella piattaforma stessa, chiamata *Java Virtual Machine*.

Java infine ha una grande comunità di sviluppatori nel mondo, proprio per il suo vasto utilizzo. Questo ha permesso lo sviluppo negli anni di moltissime librerie di codice disponibili online con licenza Open Source.

Queste librerie definiscono operazioni comuni e ricorrenti per molti domini di applicazione, permettendo quindi con facilità l'implementazione di particolari funzioni all'interno della nostra applicazione senza dover necessariamente scrivere il codice per la sua creazione, ma semplicemente riutilizzando del codice già disponibile (ovvero il principio del "riuso del codice"), facilitando il lavoro di programmazione stesso.

[6]Il bytecode è un linguaggio intermedio più astratto tra il linguaggio macchina e il linguaggio di programmazione, usato per descrivere le operazioni che costituiscono un programma. Un linguaggio intermedio come il bytecode è molto utile a coloro che realizzano linguaggi di programmazione perché riduce la dipendenza dall'hardware e facilita la creazione degli interpreti del linguaggio stesso. Fonte Wikipedia.

[7]Il JRE, o Java Runtime Environment, è un ambiente di esecuzione per applicazioni scritte in linguaggio Java, distribuito gratuitamente da Oracle. Fonte Wikipedia.

2.2.2 L'ambiente di sviluppo

Per poter scrivere codice in un determinato linguaggio di programmazione è necessario un ambiente di sviluppo all'interno del quale poter scrivere, compilare ed eseguire il suddetto codice.

Per lo sviluppo di questa applicazione è stato utilizzato *Eclipse*, un noto ambiente di sviluppo integrato (*Integrated Development Environment*, oppure semplicemente IDE) appartenente alla categoria dei software Open Source per la programmazione Java e non solo.

Questo software fornisce al programmatore tutta una serie di aiuti durante lo sviluppo del codice sorgente del programma, partendo da un editor completo di molte funzioni per la scrittura del codice, passando per la segnalazione di molti errori, ad esempio di sintassi o di semantica del codice scritto oppure nell'utilizzo di certi costrutti del linguaggio di programmazione, nonché fornendo strumenti per la fase di *debug*[8] del codice ed includendo un compilatore e/o un interprete per il linguaggio di programmazione selezionato.

La versione di Eclipse utilizzata per lo sviluppo dell'applicazione è "Luna SR2(4.4.2)".

Di questo IDE abbiamo utilizzato anche uno specifico componente aggiuntivo, *ObjectAID*, per a generazione automatica dei diagrammi UML delle classi a partire dal progetto Java sviluppato in Eclipse.

[8]Indica l'attività che consiste nell'individuazione da parte del programmatore della porzione di software affetta da errore (*bug*) rilevata nei software a seguito dell'utilizzo del programma. Fonte Wikipedia.

2.2.3 Il pattern Modified Model-View-Controller

Per lo sviluppo di questa applicazione era necessario combinare diverse funzionalità, a partire da quelle grafiche per fornire all'utente la possibilità di gestire, modificare ed analizzare i log tramite una interfaccia, passando per quelle necessarie all'elaborazione delle operazioni richieste dall'utente in maniera separata dalle componenti grafiche (che non hanno questa capacità), fino a quelle utili per la memorizzazione e gestione dei dati necessari allo svolgimento di queste elaborazioni.

Per poter combinare queste funzionalità in maniera logica e strutturata, l'applicazione è stata sviluppata utilizzando il pattern *Modified Model-View.Controller*, di cui parleremo a fondo nella Sezione 3.1.

2.2.4 Le librerie esterne

Durante lo sviluppo dell'applicazione, per l'implementazione di alcune particolari funzionalità, sono state selezionate delle librerie Java Open Source presenti online, che verranno adesso presentate.

Queste librerie sono state scelte per implementare in maniera professionale e completa alcune delle funzioni che il sistema voleva fornire all'utente, nonché per la gestione di determinati formati (primo tra tutti XES) all'interno dell'applicazione.

2.2.4.1 OpenXES

OpenXES è la libreria Java Open Source ufficiale dello standard XES, rilasciata dagli stessi sviluppatori dello standard, descritto nella Sezione 2.1.

La versione utilizzata è la 2.0, ultima release ufficiale al momento della scrittura di questo elaborato.

Per la stesura di questa sezione riguardante la libreria OpenXES abbiamo fatto riferimento a [6].

La libreria, implementazione di riferimento dello standard, è stata creata con i seguenti obiettivi:

1. Essere aderente in ogni aspetto allo standard XES.

2. Essere di immediato utilizzo e facile da integrare per gli sviluppatori che ne vogliano fare uso.

3. Fornire la massima performance per quanto riguarda la gestione e la memorizzazione dei dati dei log, qualunque sia il tipo di log preso in esame.

4. Servire da chiara e comprensibile implementazione di riferimento per altre implementazioni dello standard.

Questa libreria è stata quindi pensata per tutti gli sviluppatori che necessitino di inserire nelle loro applicazioni Java la memorizzazione, gestione, serializzazione ed analisi dei log di eventi, nel rispetto totale dello standard XES.

Nella Figura 2.4 viene mostrato il diagramma UML 2.0 delle classi della libreria che implementano i rispettivi oggetti dello standard XES.

La libreria contiene oltre a queste altre classi per l'implementazione Java di molte funzionalità connesse al rispetto dello standard e alla memorizza-

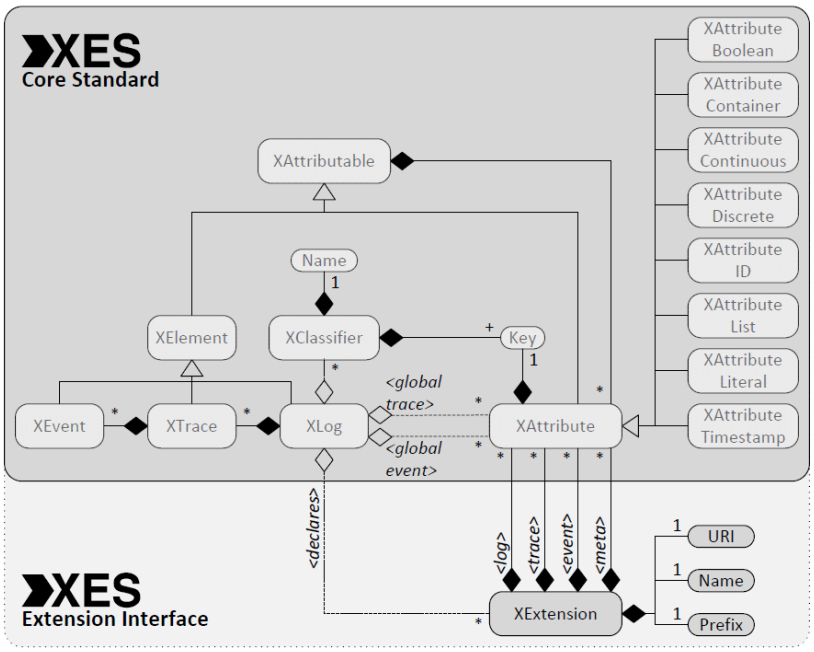

Figura 2.4: Il diagramma UML 2.0 delle classi della libreria OpenXES [6]

zione, gestione, lettura e scrittura in diversi formati dei log di eventi, ma che non vengono mostrate nel diagramma.

Delle classi presenti nel diagramma e di quelle non presenti ma incluse nella libreria ed utilizzate nello sviluppo dell'applicazione faremo menzione nel Capitolo 5, quando andremo ad analizzare le funzionalità implementate all'interno del sistema per la generazione e analisi dei log.

OpenXES costituisce uno dei pilastri più importanti dell'applicazione sviluppata, in quanto fornisce tutte le classi Java necessarie per la gestione e analisi di log di eventi presi in input sotto forma di semplici documenti testuali e trasformati grazie a questa libreria in strutture ben definite da poter utilizzare all'interno di procedure di programmazione, rispettando allo stesso tempo tutte le specifiche dello standard XES viste in precedenza.

2.2.4.2 RSyntaxTextArea

RSyntaxTextArea è una libreria Java Open Source che implementa un editor testuale per applicazioni Java, da inserire dove necessario all'interno di un qualche componente grafico (Java Swing).

La versione utilizzata all'interno dell'applicazione è la 2.5.6.

L'editor implementato da questa libreria estende le API[9] Java (in praticolare la classe `JTextComponent`), in modo da essere totalmente integrato con il pacchetto Java standard `javax.swing.text`. Questo vuol dire che l'editor è una specializzazione (o "estensione", in termini più tecnici) di quello di default dell'architettura Java e può quindi essere utilizzato al posto di quest'ultimo in maniera del tutto equivalente.

Essendo inoltre una estensione di un componente standard Java, l'editor mette a disposizione ("eredita", in termini tecnici) anche tutte le funzionalità di questo componente, oltre alle proprie, permettendo così di utilizzare sia le funzionalità già presenti nel sistema, sia quelle nuove che RSyntaxTextArea mette a disposizione.

Questa libreria mette a disposizione moltissime funzionalità da poter attivare e rendere disponibili all'interno della propria applicazione, in modo da poter usare questo editor per la gestione non solo di semplice testo, ma di veri e propri linguaggi:

1. L'evidenziamento della sintassi (*syntax highlighting*) di più di quaran-

[9] *Application Programming Interface* (in italiano Interfaccia di Programmazione di un'Applicazione): indica ogni insieme di procedure disponibili al programmatore, di solito raggruppate a formare un set di strumenti specifici per l'espletamento di un determinato compito all'interno di un certo programma. Spesso con tale termine si intendono le librerie software disponibili in un certo linguaggio di programmazione. Fonte Wikipedia.

ta linguaggi di programmazione conosciuti (tra i tanti indichiamo il linguaggio XML, quello utilizzato nella nostra applicazione) e non, visto che vi è anche la possibilità di aggiungere linguaggi personalizzati all'insieme di quelli presenti.

2. Il raggruppamento del codice (*code folding*), il bilanciamento delle parentesi (*bracket matching*) e l'indentazione automatica (*auto-indentation*) per tutti i linguaggi per i quali queste funzionalità sono definite.

3. Per i linguaggi con una sintassi ben definita (ad es. Java), la possibilità di impostare un controllore del codice (*parser*) in modo che evidenzi, mentre l'utente digita caratteri all'interno dell'editor, possibili errori di scrittura dipendenti dal linguaggio di programmazione utilizzato.

4. La gestione ottimale del caricamento e salvataggio dei documenti, nel rispetto della formattazione degli stessi in base al linguaggio di programmazione che contengono.

5. La possibilità di definire macro per registrare e riprodurre in seguito complessi insiemi di azioni all'interno dell'editor, in modo da semplificare operazioni che vengono ripetute spesso.

6. La selezione del tema da utilizzare per la colorazione della sintassi del linguaggio (a scelta tra un insieme di temi a disposizione, oppure definendone uno personale) e del font per il testo.

7. Molte altre funzionalità utili per l'editing di linguaggi, come ad esempio l'evidenziamento di tutte le occorrenze di un identificatore, variabile o funzione presente nella posizione del cursore, numeri di linea a

lato dell'editor, evidenziamento di linea, illimitata registrazione delle operazioni di "Annulla" (*Undo*) e "Ripeti" (*redo*) e molte altre ancora.

RSyntaxTextArea è stata scelta per implementare l'editor dell'applicazione dove verranno gestiti i log di eventi (e non solo, considerando che l'applicazione gestisce anche altri tipi di documento all'interno dell'editor) e per fornire all'utente moltissime proprietà utili per un più facile ed immediato lavoro su questi documenti.

Una possibile alternativa considerata: Bounce

Un'alternativa a questa libreria che era stata inizialmente presa in considerazione (prima di trovare, testare ed approvare quella poi utilizzata) era la libreria *Bounce*.

Questa libreria aveva inizialmente suscitato interesse in quanto implementava un *Editor Kit* Java da utilizzare all'interno di un'area testuale, fornendo molte funzionalità all'utente.

Tra queste vi era l'evidenziamento della sintassi del codice attraverso l'utilizzo dei colori, i numeri di linea, una funzione automatica di completamento dei tag XML e di indentazione automatica del codice.

A queste caratteristiche, che sono presenti anche nella libreria RSyntaxTextArea, si aggiungevano però una serie di problemi, tra i più importanti l'assenza di una funzione di *antialiasing* all'interno dell'editor (presente invece in RSyntaxTextArea), che rendeva quindi il testo sfocato e non ben leggibile, e il fatto che questo Editor Kit non permetteva l'accesso diretto al contenuto dell'area testuale, interponendosi come filtro tra il sistema e il testo.

Questo secondo importante problema è stato quello decisivo per il cambio di libreria da utilizzare, in quanto il nostro sistema aveva bisogno di un accesso diretto all'area testuale per gestire molte funzionalità derivate da azioni dell'utente sul testo che l'Editor Kit di fatto nascondeva all'applicazione, per gestirle autonomamente.

Dopo una attenta ricerca siamo giunti alla scoperta della libreria RSyntaxTextArea, che non solo forniva le funzionalità presenti nella libreria Bounce, ma ne forniva moltissime altre, alcune delle quali davvero interessanti come abbiamo visto precedentemente.

Inoltre questa libreria estende le API Java fornendo in questo modo delle classi che sono una specializzazione di quelle standard di Java, lasciando quindi al sistema libero accesso all'area testuale e libera gestione del suo contenuto.

La scelta di cambiare libreria per l'implementazione dell'editor dell'applicazione è stata dunque relativamente facile, a seguito delle considerazioni appena fatte a riguardo.

2.2.4.3 AutoComplete

Oltre alle funzionalità messe a disposizione di default dalla libreria RSyntaxTextArea, presentata nella precedente sezione, è possibile attivarne altre all'interno dell'editor grazie alla libreria *AutoComplete*, una estensione di RSyntaxTextArea che aggiunge appunto ulteriori proprietà a quest'ultima (*add-on library*).

Questa libreria permette infatti di registrare una serie di porzioni di codice (*code snippets*) e di associare ad essi un identificatore, una qualsiasi parola

chiave che rappresenti quella determinata componente.

Una volta registrati, è possibile richiamare questi blocchi di codice all'interno dell'editor scrivendo l'identificatore associato alla porzione di codice scelta nel punto dove vogliamo che appaia e premendo la combinazione di tasti Ctrl + Shift + Spazio sulla tastiera.

Questo permette di scrivere codice molto velocemente, richiamando blocchi precedentemente preparati per essere usati quando necessario, riducendo i tempi di programmazione e diminuendo notevolmente il rischio di errori.

Oltre alle porzioni di codice, AutoComplete permette la registrazione di un elenco di parole, ad esempio elementi della sintassi di un linguaggio, per andare a formare una sorta di vocabolario di termini.

Quando all'interno dell'editor si inizia a scrivere una parola e poi si preme il comando Ctrl + Spazio sulla tastiera (la funzionalità è tuttavia attivabile anche in maniera automatica), apparirà un elenco dei possibili termini tra cui scegliere che iniziano con quella porzione di testo e che fanno parte della lista degli elementi registrati.

In questo modo si potrà avere all'interno dell'editor un sistema di completamento automatico del codice (*code completion*), per fornire all'utente i termini corretti della sintassi del linguaggio selezionato ad esempio, oppure possibili identificatori, valori o attributi definiti per determinate situazioni.

La scrittura del codice risulta ancora più rapida e semplice aggiungendo questa libreria a quella della precedente sezione, mettendo l'utente nelle condizioni di non dover conoscere a memoria la sintassi del linguaggio per poter utilizzare l'applicazione.

E' utile notare come sia possibile premere il comando Ctrl + Spazio su

un qualsiasi punto del codice all'interno dell'editor, per far apparire l'intera lista delle parole disponibili tra cui selezionare, senza quindi dover per forza ricordare l'inizio di una di esse.

2.2.4.4 FlyingSaucer

FlyingSaucer è una libreria Java Open Source per la conversione di documenti HTML in formato PDF.

La libreria fa parte del progetti presenti su Google Code[10] ed è sviluppata da membri di questa comunità.

FlyingSaucer prende in input un documento XML o XHTML ed elaborando i suoi tag ed il CSS associato (presente all'interno del documento), genera in output un documento PDF del tutto fedele alla rappresentazione visuale del documento originario.

Questa libreria è stata scelta per le sue ottime performance applicate agli scopi di questa applicazione, ovvero la conversione di documenti XHTML con CSS integrato in documenti PDF che siano una quanto più accurata rappresentazione dell'originale, con buone performance in termini di tempi di elaborazione e qualità del prodotto finale, nonché di facilità di integrazione con il sistema sviluppato.

2.2.4.5 HTMLtoLaTeX

HTMLtoLaTeX è un programma Java Open Source per la conversione di documenti HTML in formato LaTeX.

[10] https://code.google.com/p/flying-saucer/

Il progetto è distribuito all'interno del database di *SourceForge*[11] ed è stato sviluppato da Michal Kebrt.

Questo progetto, fermo dal 2010 alla release 1.0.1, in realtà è una piccola applicazione indipendente che permette la selezione di un documento HTML e restituisce come output un documento in formato LaTeX.

Per l'applicazione in esame però era sufficiente una libreria contenente le classi che implementavano questa procedura (e quindi era necessario solamente l'accesso alle classi di questa applicazione che implementavano la funzione di conversione, e non a quelle per la grafica e l'avvio della stessa).

Il progetto dal 2010 non è mai stato convertito o fornito sotto forma di libreria inseribile in altre applicazioni, quindi è stato necessario effettuare prima questa conversione del progetto in libreria, per poi poterlo includere sotto questa forma nel sistema ed utilizzarne le procedure di conversione.

La scelta di questa componente è stata fatta per le numerose funzionalità e configurazioni messe a disposizione che, una volta impostate, permettono una conversione ottimale da HTML a LaTeX.

Il pacchetto infatti permette, attraverso un file testuale di configurazione, di definire, per ogni elemento del codice HTML, il corrispettivo elemento LaTeX, in modo tale che la conversione avvenga poi in maniera del tutto elementare, analizzando il documento HTML dall'alto in basso e convertendo i tag in elementi LaTeX.

Tra le tante funzionalità degne di nota, segnaliamo la gestione anche del codice CSS[12] presente all'interno del documento (che viene convertito nei

[11] http://htmltolatex.sourceforge.net

[12] *Cascading Style Sheets*, in italiano fogli di stile, è un linguaggio utilizzato per definire la formattazione di documenti HTML, XHTML e XML.

corrispettivi comandi LaTeX per la formattazione), oltre a quello HTML già citato, dei commenti HTML (convertiti in commenti LaTeX) e dei collegamenti ipertestuali (convertiti in note a piè di pagina ad esempio, oppure in elementi della bibliografia, persino in collegamenti ipertestuali del documento PDF che verrà generato dal codice LaTeX, dipende dalla configurazione scelta).

2.2.4.6 OpenCSV

OpenCSV è una semplice libreria Java Open Source contenente un *parser* per documenti in formato CSV[13].

Il progetto fa parte del database di *SourceForge*[14] e contribuiscono al suo sviluppo diversi sviluppatori della comunità di SourceForge stesso.

La versione utilizzata per la nostra applicazione è la 3.3.

La libreria mette a disposizione un sistema per la lettura e scrittura di documenti in formato CSV, con la gestione di situazioni tipiche di questo formato come ad esempio un qualsiasi numero di valori per linea, valori singoli contenuti tra separatori e valori composti contenuti tra caratteri di raggruppamento (in modo da considerare eventuali separatori utilizzati dentro gli apici come semplici valori ed eventuali ritorni a capo di linea sempre come facenti parte dello stesso elemento).

Fornisce inoltre la possibilità di definire i propri caratteri di separazione e di raggruppamento (oppure utilizzare quelli di default, rispettivamente la virgola e i doppi apici) e di leggere e scrivere il documento attraverso appositi

[13] *Comma-Separated Values.*
[14] http://opencsv.sourceforge.net

oggetti Java per poter rispettivamente leggere con un solo accesso l'intero documento e poi processare successivamente i suoi elementi (che corrispondono ciascuno ad una riga) iterando tra di essi, e scrivere semplicemente inviando insiemi di elementi che verranno scritti nel formato CSV in automatico dalla libreria, utilizzando i caratteri di separazione e raggruppamento impostati.

La libreria è stata scelta per gestire all'interno dell'applicazione documenti in formato CSV, non per quanto riguarda la loro apertura e modifica (che riguarda operazioni su semplici documenti di testo e non sul formato CSV), ma per la creazione di una funzione di trasformazione da documenti CSV a log di eventi in formato XES e viceversa.

Come vedremo meglio in dettaglio nel Capitolo 5, l'applicazione permette, a partire da un documento CSV, di generare un log di eventi in formato XES corrispondente ai dati presenti nel documento in input, e a partire da un log di eventi XES, di generare un documento CSV rappresentante i dati del log sotto forma di righe e valori tipici di questo formato.

Per l'implementazione di questa funzionalità, questa libreria ha permesso una facile lettura dei dati del documento CSV in input (ed una conseguente facile elaborazione degli stessi per la generazione del log) ed una altrettanto facile scrittura di un documento testuale in formato CSV in output (considerando che è la libreria che gestisce il rispetto del formato CSV, a partire da semplici insiemi di dati che gli vengono passati per essere scritti su una linea del documento).

Capitolo 3

PROGETTAZIONE DELL'APPLICAZIONE

In questo capitolo verranno mostrate le decisioni prese in merito alla progettazione dell'applicazione, alla sua organizzazione interna e a come le varie componenti sviluppate interagiscano tra loro.

3.1 Pattern architetturale

La prima fase della progettazione consiste nello scegliere il pattern architetturale dell'applicazione e di conseguenza decidere come organizzare il del codice e le interconnessioni tra le componenti del sistema.

La scelta è ricaduta su una generalizzazione del famoso pattern architetturale Model-View-Controller (MVC), molto diffusa nello sviluppo di siste-

mi software, in particolare nell'ambito della programmazione orientata agli oggetti.

3.1.1 Il pattern Model-View-Controller tradizionale

Il pattern Model-View-Controller tradizionale consiste nel separare i componenti software che implementano il modello delle funzionalità di business (*Model*), dai componenti che implementano la logica di presentazione (*View*) e da quelli di controllo che tali funzionalità utilizzano (*Controller*).

Utilizzando questo pattern, l'applicazione viene divisa concettualmente nelle tre componenti citate, mostrate in Figura 3.1.

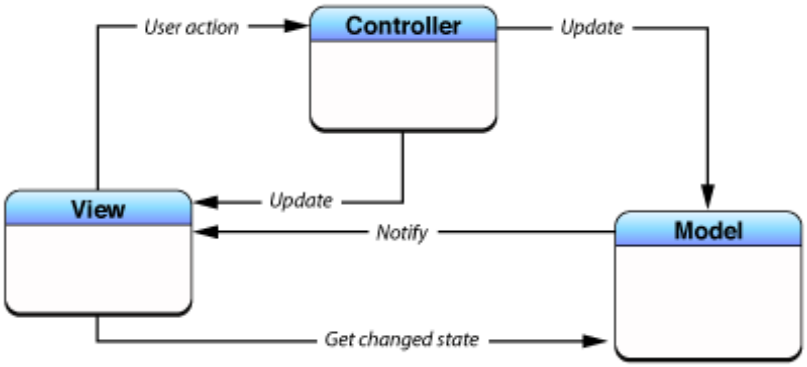

Figura 3.1: L'architettura tradizionale del Model-View-Controller [4]

Model

Il *Model*, o modello, implementa l'interfaccia per accedere ai dati, fornendo i metodi necessari per leggere, scrivere e modificare questi ultimi.

In base al tipo di applicazione questi dati possono risiedere in strutture diverse: in un database ad esempio, come nel caso di una applicazione web

o applicazioni con accesso a grandi quantità di dati registrati, oppure in file testuali, nel caso di file di configurazione, e così via.

I dati sono così nascosti alle altre componenti dell'applicazione e ogni lettura o modifica effettuata deve passare per il modello, che si accerterà di eseguire queste operazioni in maniera robusta, controllando l'esito dell'operazione ed assicurando l'integrità dei dati.

View

La *View*, o vista, implementa l'interfaccia che viene visualizzata all'utente (*Graphic User Interface*, GUI) e attraverso la quale quest'ultimo interagisce con il sistema.

La vista nel pattern MVC definisce come i dati presenti nel modello debbano essere visualizzati ed interagisce direttamente con esso per recuperarli attraverso diverse metodologie.

Una prima tecnica è l'ascolto diretto da parte della vista delle modifiche effettuate dal modello sui dati. In conseguenza di queste azioni del modello, la vista effettua un aggiornamento dei dati mostrati, uniformandosi ai valori aggiornati dal modello. In questo caso si parla di *Active Model MVC Pattern*, che è il caso mostrato in Fig. 3.1:

Una seconda tecnica è l'utilizzo di sistemi di notifica da parte del Controller alla vista che i dati sono cambiati e quindi la vista deve aggiornarsi recuperando personalmente i dati dal modello. E' il caso del *Passive Model MVC Pattern*, che rispetto al pattern mostrato in Fig. 3.1 manca dell'arco "Notify", in quanto non vi è notifica da parte del modello che i dati sono cambiati, ma questa arriva dal Controller.

La vista inoltre riceve comandi dall'utente (attraverso pulsanti, click del mouse in determinate aree, interazioni con i componenti grafici in generale) e li invia al Controller che elaborerà la richiesta dell'utente richiedendo se necessario al modello delle modifiche ai dati. Al termine delle operazioni, il Controller notificherà la vista dei cambiamenti effettuati ai dati e come questa deve modificarsi di conseguenza.

Controller

Il *Controller* è il componente che viene posizionato nel mezzo tra la vista e il modello: riceve dalla vista le azioni che l'utente compie all'interno dell'interfaccia grafica, le elabora, interagisce con il modello per recuperare o modificare i dati se necessario, dopodiché se necessario (in base alla formula "active" o "passive" del modello) comunica alla vista il risultato delle operazioni svolte e come questa debba modificarsi in conseguenza di queste ultime. E' il centro dell'applicazione, la parte che interpreta le scelte dell'utente e ne implementa la relativa operazione da svolgere, interagendo sia con il modello che con la vista.

3.1.2 Il Modified Model-View-Controller

La generalizzazione di questo pattern scelta per la nostra applicazione è mostrata in Figura 3.2

Come si vede dalla Figura 3.2, la differenza tra questa versione del pattern e quella tradizionale sta nella assenza di comunicazione diretta tra la View e il Model.

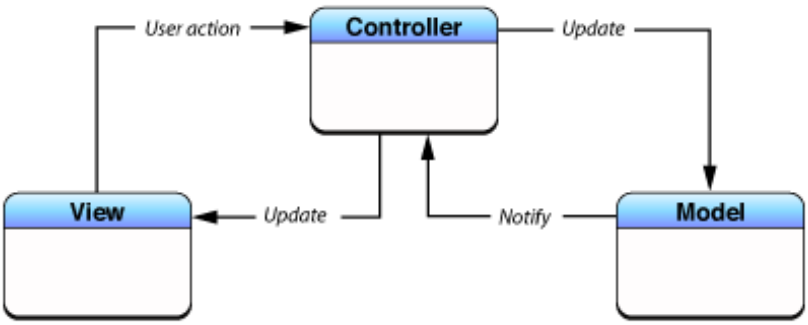

Figura 3.2: Modified Model-View-Controller [4]

In questo caso infatti, la vista non "ascolta" le modifiche che vengono fatte dal modello ai dati (richieste dal Controller) per poi aggiornarsi automaticamente di conseguenza (come nel caso sopra citato dell'Active Model MVC Pattern), né riceve comunicazione da parte del Controller che i dati sono stati modificati e quindi deve recuperare i dati aggiornati direttamente dal modello per poi aggiornarsi (come nel caso del Passive Model MVC Pattern).

La vista è all'oscuro del modello, come il modello lo è della vista. Solo il Controller ha un riferimento ad entrambe le componenti: è proprio quest'ultimo che si occupa dell'intera interazione tra di loro, in maniera esclusiva.

Il Controller riceve dalla vista le azioni compiute dall'utente all'interno dell'interfaccia grafica, le elabora, interagisce se necessario con il modello per recuperare o modificare i dati, prepara la risposta e la visualizzazione corretta dei dati ricavati e comunica alla vista come aggiornarsi in base alle operazioni svolte, indicandole esattamente il suo comportamento grafico in risposta ai comandi dell'utente.

Come si evince dai passaggi elencati, la vista non ha più bisogno di controllare eventuali modifiche ai dati da parte del modello, né andare a leggere le modifiche effettuate dal Controller una volta che questo le comunica l'operazione appena svolta: è direttamente il Controller a leggere i dati dal modello e preparare la configurazione corretta per la vista, che a questo punto dovrà semplicemente mostrare all'utente quello che il Controller ha deciso.

La scelta di questa generalizzazione del pattern MVC deriva innanzitutto dal suo largo utilizzo nello sviluppo di applicazioni con linguaggi di programmazione orientati agli oggetti: questa versione è infatti più recente rispetto a quella tradizionale e quindi più adatta allo sviluppo di sistemi che utilizzano tecnologie attuali.

Inoltre, questa scelta deriva dal fatto che la netta divisione delle tre componenti, con il Controller unico elemento di collegamento e gestione delle interazioni e View e Model che non conoscono l'esistenza l'uno dell'altro, permette non solo una facile manutenzione del codice e test del progetto sviluppato, considerando che tutta la parte di elaborazione è demandata al Controller e che la vista ha l'unico compito di rendere graficamente i contenuti che il Controller gli invia già pronti, ma anche la possibilità di cambiare una delle tre componenti mantenendo le altre invariate, vista la loro interazione lineare (passando attraverso il Controller) e non più triangolare.

Questa generalizzazione del pattern MVC tradizionale viene chiamata in diversi modi a seconda della fonte consultata: il sito di Oracle chiama questa versione "*Modified MVC*" [5], mentre il sito di Apple lo definisce "*Cocoa MVC*" [4], in quanto è la generalizzazione del pattern MVC utilizzata nel

framework Apple Cocoa [1].

Molte altre fonti invece si riferiscono a questo pattern con un nome diverso, per sottolineare le differenze con il modello tradizionale, quello di *Model-View-Presenter (MVP): Passive View.*

Il nome "Model-View-Presenter" indica una elaborazione del pattern Model-View-Controller dove vi è una impostazione diversa riguardo a quale componente dei tre gestisce le differenti elaborazioni del sistema.

La versione "Passive View" invece indica una specifica variante del pattern MVP dove la vista è delegata semplicemente della resa grafica dei contenuti che il Presenter le invia, dopo aver elaborato il comando dell'utente ed aver interagito con il modello, e che in pratica indica esattamente il funzionamento del pattern da noi utilizzato e chiamato con nomi diversi da altre fonti.

Nel seguito di questo documento, per semplicità di esposizione, indicheremo le tre componenti del sistema utilizzando i termini Model (o modello), View (o vista) e Controller, riferendoci con questi nomi al pattern generalizzato scelto e appena descritto in questa sezione.

3.2 Struttura dei pacchetti Java

Le classi Java che costituiscono l'applicazione sono state suddivise in pacchetti (*packages*), per organizzarne la struttura seguendo il pattern scelto.

[1] Cocoa è una API orientata agli oggetti nativa di Apple per il sistema operativo OS X. Esiste anche una versione di queste API per il sistema operativo per dispositivi mobili iOS, chiamata *Cocoa Touch.* Fonte Wikipedia.

L'albero dei pacchetti è inizialmente costituito da una serie di nodi, che definiscono via via più dettagliatamente la struttura dell'applicazione:

1. `unipi`: indica l'università all'interno della quale è stata sviluppata questa applicazione, ovvero l'Università di Pisa.

2. `bpm`: indica il corso universitario le cui tematiche fanno da background a questa applicazione, ovvero il corso di *Business Process Modeling*.

3. `elm`: indica il nome dell'applicazione, *Event Log Manager*.

4. `santeramo`: indica il cognome dello studente che ha sviluppato il codice che segue questo pacchetto, in questo caso Santeramo Francesco.

Questa definizione porta ad avere un albero iniziale dei pacchetti della forma `unipi.bpm.elm.santeramo`, che prosegue poi con i pacchetti che implementano l'applicazione (tenendo sempre presente che la radice di questo albero è la cartella che contiene i sorgenti dell'applicazione, che non viene riportata nella sequenza dei pacchetti ma che di fatto fa da radice dell'albero).

Questa struttura serve da base per future estensioni dell'applicazione, in base alla loro provenienza: una estensione da parte di un'altra università, si affiancherà al pacchetto `unipi` con un altro sotto-albero simile a quello descritto; una estensione interna all'Università di Pisa, ma fatta da un altro corso universitario, verrà identificata dal pacchetto `unipi` come pacchetto iniziale e poi affiancherà il preesistente `bpm` con un sotto-albero simile a quello presente; allo stesso modo un progetto interno al corso di Business Process Modeling che si affianca come estensione a questo, sarà inserito nel pacchetto `unipi.bpm` con nome dell'estensione creata.

All'interno dei nodi indicati, in particolare all'interno del pacchetto `santeramo`, troviamo il codice dell'applicazione, suddiviso a sua volta in una serie di pacchetti che ne definiscono la struttura secondo il pattern Model-View-Controller.

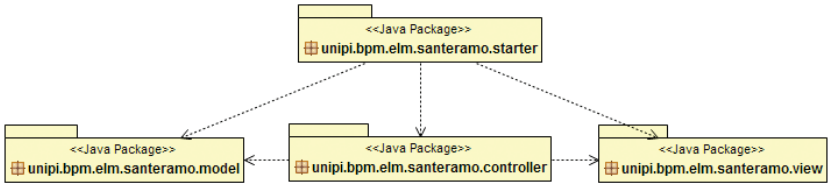

Figura 3.3: Diagramma UML dei pacchetti dell'applicazione

In Figura 3.3 sono mostrati i pacchetti che definiscono tale struttura. Come si può facilmente notare, la struttura è estremamente semplice.

Il pacchetto `starter` contiene una sola classe, quella che si preoccupa di creare la View, il Model ed il Controller, passando a quest'ultimo il controllo dei primi due componenti ed infine chiedendogli di avviare l'applicazione.

I restanti tre pacchetti, `model`, `view` e `controller`, sono la chiara suddivisione delle classi Java nelle tre componenti del pattern MVC: questi pacchetti contengono tutte le classi che implementano le funzionalità rispettivamente della View, del Model e del Controller.

3.3 Diagrammi UML delle classi

In questa sezione verranno mostrati e descritti i diagrammi UML dei pacchetti che definiscono la struttura dell'applicazione e che contengono le varie classi Java.

I diagrammi UML sono stati creati utilizzando il componente (*plugin*) *ObjectAid UML Explorer* dell'IDE Eclipse, presentato nella Sez. 2.2.2 del Cap. 2.

3.3.1 Il pacchetto Starter

Il pacchetto `starter` contiene una sola classe, chiamata `EventLogManagerStarter`, che, dopo aver controllato quale sia il sistema operativo sottostante, sceglie la relativa grafica da utilizzare; dopodiché crea vista, modello e Controller, associando a quest'ultimo le altre due componenti e demandandogli l'avvio dell'intero sistema.

Non mostreremo nello specifico il suo diagramma UML, essendo appunto composto da una sola classe, ma terremo conto di quest'ultima nella Sezione 3.3.5 in quanto elemento di avvio dell'intera applicazione Java.

3.3.2 Il pacchetto Model

Il pacchetto `model` contiene la classe `EventLogManagerModel`, che rappresenta la classe principale del pacchetto, e la classe annidata (*inner class*) `MostRecentElementList<E>`, una classe utilità a disposizione della principale per la gestione di liste di tipo LIFO (*Last In First Out*) a capacità limitata.

La classe `EventLogManagerModel` come detto è la principale all'interno del pacchetto ed implementa l'interfaccia di comunicazione tra il pacchetto `model` e gli altri del pattern MVC, fornendo al Controller tutti i metodi necessari per la gestione e modifica dei dati memorizzati dal sistema.

In Figura 3.4 viene riportato il diagramma UML delle classi, contenente la classe principale e la classe annidata.

La relazione della classe annidata con la principale è indicata da un lato dall'arco che termina con un cerchio crociato e che indica appunto che la seconda, `MostRecentElementList<E>`, è una classe interna della principale.

Dall'altro, questa relazione è sottolineata da un arco a freccia che indica l'associazione tra le due effettuata attraverso un riferimento (`recentLogList`) della seconda all'interno della principale.

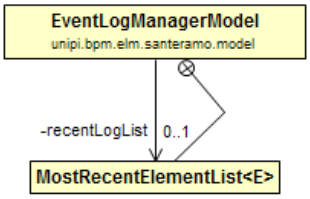

Figura 3.4: Diagramma UML del pacchetto Model

3.3.3 Il pacchetto View

Il pacchetto `view` è costituito dalla classe `EventLogManagerView`, classe principale del pacchetto che rappresenta la componente della vista nel pattern MVC, e da una serie di classi che definiscono i vari componenti della vista ed il loro comportamento.

Per quanto riguarda la classe `EventLogManagerView`, questa fornisce tutti i metodi per permettere al Controller di "ascoltare" le azioni dell'utente (ovvero, in termini tecnici, di installare dei *listener* sulle componenti della vista per poter monitorare gli eventi che accadono su di essi a partire dalle

azioni dell'utente) e per informare la vista stessa delle elaborazioni effettuate e come di conseguenza questa debba modificarsi.

E' di fatto la classe che implementa la componente della View e che permette a questo pacchetto di interagire con gli altri.

Le altre classi di questo pacchetto implementano invece una serie di componenti grafici della vista. Si tratta di oggetti speciali utilizzati nell'interfaccia grafica con determinate specifiche a seconda del loro compito, e di un insieme di componenti manager e layout per la visualizzazione dei componenti grafici. Queste classi vengono utilizzate dalla classe principale per la visualizzazione di una interfaccia grafica specifica per l'applicazione.

Nella Figura 3.5 viene mostrato il diagramma UML di questo pacchetto.

Al centro vi è la classe principale che permette la comunicazione con gli altri pacchetti: `EventLogManagerView`.

Intorno ad essa le altre classi che forniscono a quest'ultima componenti grafiche specifiche e componenti speciali per la gestione di questi elementi.

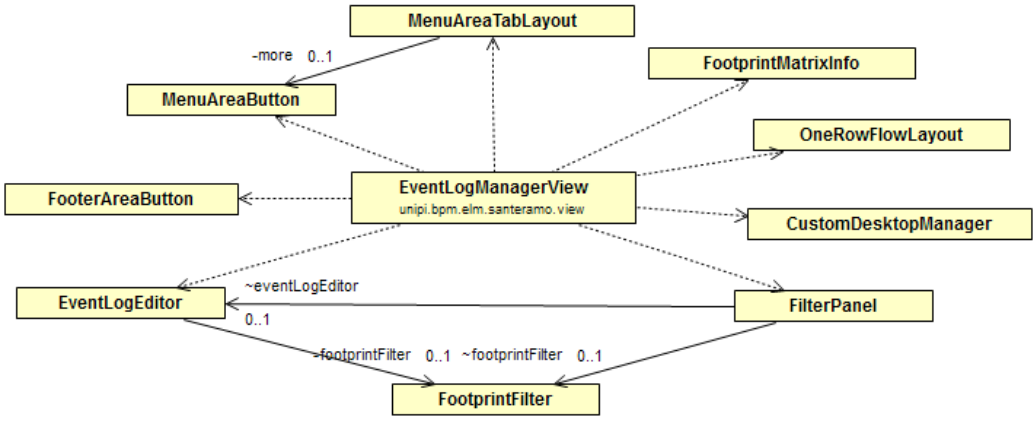

Figura 3.5: Diagramma UML del pacchetto View

Andiamo adesso a descrivere brevemente le classi che sono in dipendenza con la classe principale. In questo capitolo ci limiteremo a dare una definizione più astratta del loro utilizzo e comportamento, rimandando ai successivi una descrizione più dettagliata.

Le classi sono le seguenti:

1. `EventLogEditor`: questa classe definisce la struttura di una particolare finestra interna del sistema (tecnicamente, un frame interno al frame principale dell'applicazione, detto anche *internal frame*) che viene utilizzata per la visualizzazione dei log di eventi e per la scrittura e modifica del loro codice in linguaggio XES. É di fatto un editor testuale e viene utilizzato dall'applicazione anche per la modifica di qualsiasi documento di testo che venga aperto in essa. Ha numerose proprietà che vedremo nei successivi capitoli di questo documento.

2. `FootprintFilter`: ogni `EventLogEditor` contenente un log ha associato uno specifico filtro per quel log che verrà utilizzato durante la fase di analisi per ottenere risultati più accurati. Un oggetto di questa classe è associato a ciascun `EventLogEditor` per registrare i filtri attualmente impostati dall'utente per quello specifico log di eventi.

3. `FilterPanel`: questa classe definisce un pannello mobile dell'applicazione contenente i filtri del log selezionato, in modo da mostrarli all'utente e permettergli di modificarli e cambiare così l'esito dell'analisi. Ha una associazione sia con l'`EventLogEditor` alla quale appartiene, sia con i filtri definiti per il log che questo editor contiene (filtri definiti attraverso la classe `FootprintFilter`) per potervi accedere diretta-

mente per le operazioni di lettura e modifica. Anche se non mostrato nel diagramma, esiste una associazione anche dalla classe `EventLogEditor` e questa classe, creando di fatto una associazione bidirezionale. Sia l'editor che il pannello dei filtri infatti hanno un riferimento all'altra componente, questo perché una azione su una delle due da parte dell'utente deve risultare in una modifica all'altra componente, quindi queste due devono necessariamente essere associate tra loro. Il motivo per il quale questa associazione non è visualizzata nel diagramma è che, mentre la classe `FilterPanel` ha un riferimento ad un oggetto `EventLogEditor` (chiamato `eventLogEditor` nel diagramma in Fig. 3.5), la classe `EventLogEditor` ha un riferimento ad un generico oggetto grafico (tecnicamente, ad un oggetto di tipo *Dialog* il cui riferimento è chiamato `filterDialog` nel diagramma in Fig.3.5) per potersi associare a qualsiasi tipo di pannello e non solo ad un `FilterPanel`. Ne consegue che l'associazione c'è, ma non è diretta e quindi non viene visualizzata nel diagramma.

4. `FootprintMatrixInfo`: questa classe definisce la struttura di una particolare finestra interna del sistema (come fa `EventLogEditor`) utilizzata per la visualizzazione di codice XHTML. Il codice XHTML è utilizzato dal sistema per la visualizzazione grafica dei risultati dell'analisi sui log, ovvero una matrice delle dipendenze causali ed una serie di informazioni che vedremo in dettaglio nei capitoli successivi.

5. `CustomDesktopManager`: per la gestione all'interno della finestra principale del sistema delle finestre interne sopra citate, viene utilizzato

questo manager, che si preoccupa di disporre ogni nuova finestra che viene aperta ed inserita all'interno della principale in uno spazio libero (se disponibile), evitando la sovrapposizione totale delle finestre tramite uno stile di disposizione detto "a cascata".

6. `MenuAreaButton`: questa classe definisce la struttura e la grafica dei pulsanti visualizzati nel menù principale dell'applicazione. Questi pulsanti sono statici, ovvero vengono definiti dall'applicazione al suo avvio e mostrati all'utente nel menù.

7. `MenuAreaTabLayout`: la disposizione dei pulsanti all'interno del menù principale è definita da questo layout. In particolare, nel momento in cui non c'è abbastanza spazio nella finestra per mostrare tutti i pulsanti disponibili, questo layout li riorganizza inserendone quanti più possibile nel menù in alto e utilizzando in fondo a destra un pulsante "More" per unire in un sotto-menù tutti quelli che non hanno spazio a disposizione. Nel diagramma questo layout ha una associazione con la classe `MenuAreaButton`, in quanto, sebbene questo layout gestisca dinamicamente un qualsiasi numero di pulsanti, ne utilizza uno in particolare in maniera fissa (il pulsante "More" appunto, indicato dal riferimento `more`) in caso di necessità.

8. `FooterAreaButton`: questo classe definisce la struttura e la grafica dei pulsanti visualizzati nel menù in fondo alla finestra principale. Questi pulsanti, a differenza dei `MenuAreaButton`, sono dinamici, ovvero ne viene inserito uno per ogni finestra che viene aperta all'interno dell'applicazione, riportante in esso il nome della finestra ad esso collegata,

per permetterne la gestione (operazioni classiche come la minimizzazione o massimizzazione della finestra, la selezione e la chiusura della stessa). Questi pulsanti vengono poi dinamicamente rimossi quando la finestra alla quale sono associati viene chiusa.

9. `OneRowFlowLayout`: i pulsanti inseriti nella parte in fondo alla finestra principale, i `FooterAreaButton`, possono essere in numero variabile. Se ne viene inserito un numero maggiore rispetto allo spazio disponibile per visualizzarli tutti, questo layout si preoccupa di ricalcolare la dimensione di tutti quelli presenti e diminuirla (o aumentarla, se in seguito ad un precedente ridimensionamento, nel caso qualcuno di essi venga chiuso e si renda disponibile più spazio per la visualizzazione) proporzionalmente in modo da visualizzarli tutti nello spazio disponibile.

3.3.4 Il pacchetto Controller

Il pacchetto `controller` contiene tutte le classi che implementano il comportamento della componente Controller del pattern MVC.

Questo pacchetto è composto da numerosi classi, di cui `EventLogManagerController` ne è la principale: è la classe che rappresenta appunto il Controller e permette l'interazione tra la vista ed il modello..

Per ogni operazione che l'utente può eseguire nell'interfaccia grafica della View, questa classe contiene un metodo per la sua elaborazione. Utilizzando i riferimenti alle classi `EventLogManagerModel` e `EventLogManagerView` presentate nelle precedenti sezioni (e resi espliciti nel diagramma UML del-

le classi di questo pacchetto, per sottolineare la funzione del Controller di gestore delle interazioni tra le componenti), questa classe recupera ed eventualmente modifica i dati necessari e comunica alla vista come questa debba modificarsi in conseguenza di tale elaborazione.

In Figura 3.6 viene mostrato il diagramma UML di questo pacchetto, contenente al centro la classe principale appena descritta e intorno tutte le classi da essa utilizzate per assolvere al suo compito di Controller.

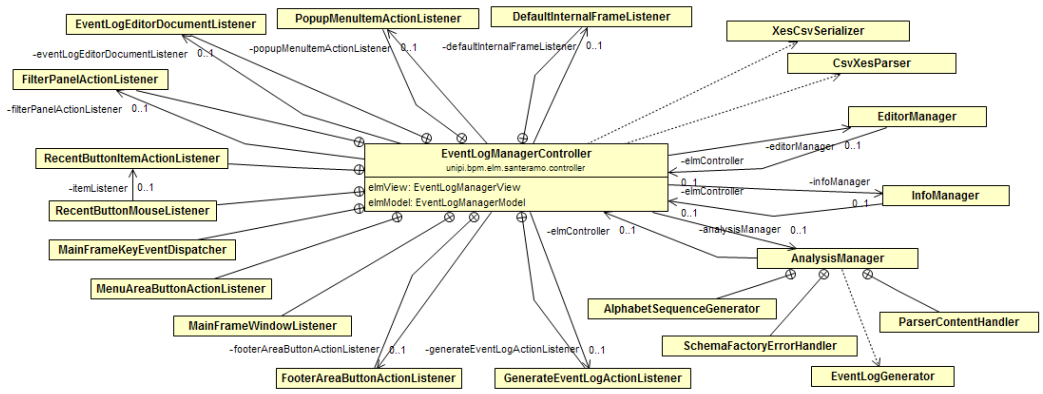

Figura 3.6: Diagramma UML del pacchetto Controller

Le classi mostrate nel diagramma intorno alla classe principale, `EventLogManagerController`, possono essere suddivise in alcune categorie, per poterne spiegare più facilmente il comportamento ed utilizzo.

3.3.4.1 Le classi Listener

Le classi *listener* non sono altro che degli oggetti che vengono associati ad elementi della vista e permettono al Controller di ascoltare il comportamento di questi oggetti in risposta alle azioni dell'utente su di essi.

In questo modo il Controller "riceve" dalla vista i comandi dell'utente e può elaborarne la relativa operazione.

Tutte le classi listener sono classi interne (*inner classes*) della classe principale del Controller, come indicato nel diagramma dagli archi che terminano con il cerchio crociato.

Alcune di esse vengono utilizzate una sola volta dal Controller, associandole in fase di avvio dell'applicazione ad esempio ad un qualche componente statico della vista, e quindi non vi è una associazione a queste ultime con la classe principale, in quanto non necessario.

Altre invece vengono utilizzate durante l'esecuzione dell'applicazione per essere associate a componenti generati ed inseriti dinamicamente all'interno dell'interfaccia grafica e per le quali quindi è necessario avere un rifermento da parte del Controller, in modo da poterle riutilizzare quando necessario. Queste classi sono indicate dall'arco a freccia che indica appunto questa associazione ed il nome del riferimento.

I nomi di questi listener sono la composizione di due parti: la prima metà è il nome identificativo del listener, per indicare per quale componente grafica è stato creato; la seconda metà richiama il nome del suo tipo (tecnicamente, l'interfaccia che implementa o la classe che estende, a seconda dei casi), in modo da rendere evidente già dal nome di che tipo di listener si tratti.

Vediamo quali sono le classi listener del pacchetto `controller`:

1. `MainFrameWindowListener`: questo listener ascolta le azioni eseguite sulla finestra principale (*main frame*), come ad esempio quando l'utente preme il tasto di chiusura della finestra, in modo che il sistema possa

controllare se ci sono dei documenti non salvati e chiedere se si vogliano salvare prima di chiudere l'applicazione.

2. `MenuAreaButtonActionListener`: questa classe sta in ascolto sui pulsanti del menù principale, in alto nella finestra. Quando uno di questi pulsanti viene premuto, questo listener chiama il metodo del Controller associato a quel pulsante per elaborare il comando e restituire la risposta all'utente.

3. `RecentButtonMouseListener`: questo listener è specifico per un pulsante della vista, il pulsante "Recent". Quando si preme questo pulsante infatti, quello che si vuole è aprire un pop-up con una lista degli ultimi documenti aperti, in ordine dal più recente al meno recente, con la possibilità di sceglierne uno da aprire. Quello che fa questa classe è proprio generare un pop-up alla pressione del pulsante Recent, popolandolo con i dati corretti e demandando poi a un altro listener (`RecentButtonItemActionListener`, associato a questa classe tramite il riferimento `itemListener`) l'ascolto sugli elementi di questa lista.

4. `RecentButtonItemActionListener`: questo listener è quello di cui parlavamo al punto tre, ovvero quello che sta in ascolto sugli elementi del pop-up aperto in attesa di una scelta da parte dell'utente. Una volta che l'utente sceglie uno degli elementi, questa classe si premura di compiere tutte le operazioni necessarie all'apertura del documento.

5. `DefaultInternalFrameListener`: è il listener che gestisce il comportamento delle finestre interne quando l'utente interagisce con loro (se-

lezione, minimizzazione, riduzione e chiusura). Ad ognuna di queste operazioni il listener risponde con una modifica specifica dell'aspetto della finestra e, in caso di chiusura della stessa, con la cancellazione dei riferimenti ad essa collegati (è il caso del `FooterAreaButton` e del `FilterPanel` associati ad una finestra interna dell'applicazione) e con eventuali messaggi di salvataggio o esportazione del contenuto, in caso risulti che questa procedura non sia già stata fatta.

6. `EventLogEditorDocumentListener`: questa classe è un particolare listener che controlla ogni azione eseguita all'interno di un editor a cui viene associato. Nello specifico, quando l'utente inserisce, rimuove o modifica del testo all'interno di un `EventLogEditor`, questo listener "cattura" questa operazione e modifica lo stato dell'editor di conseguenza, per indicare che il contenuto in esso presente è stato modificato rispetto all'ultima versione salvata.

7. `FilterPanelActionListener`: quando l'utente decide di visualizzare il pannello dei filtri di un editor, non ha solo la possibilità di visionare i filtri impostati, ma anche di modificarli e di eseguire analisi direttamente dal pannello stesso. Questo listener resta in ascolto delle modifiche ai filtri che l'utente compie e di quale operazione quest'ultimo decida di avviare successivamente, in modo da implementare la corretta operazione da svolgere in base alla scelta fatta.

8. `GenerateEventLogActionListener`: in maniera simile al precedente listener, questa classe gestisce le scelte dell'utente all'interno del pannello per la generazione di log di eventi a partire da espressioni testuali.

9. `FooterAreaButtonActionListener`: questo listener resta in ascolto delle azioni dell'utente eseguite sui pulsanti presenti nella barra in fondo alla finestra principale (i pulsanti `FooterAreaButton`, descritti nella Sez. 3.3.3). Le azioni possibili si riconducono ad una, un semplice click su di essi, ma in base alla condizione in cui si trova la finestra interna associata al pulsante selezionato cambia l'azione che viene eseguita dal sistema. I pulsanti in basso gestiscono lo stato delle finestre interne, selezionandole se non lo sono, minimizzandole se selezionate e massimizzate, massimizzandole viceversa.

10. `PopupMenuItemActionListener`: se un utente decide di premere il tasto destro del mouse su un pulsante `FooterAreaButton`, apparirà un pop-up con delle opzioni da selezionare. Questa classe è la classe adibita alla gestione di quelle opzioni, definendo per ciascuna di esse il comando da eseguire e restando in attesa che l'utente ne selezioni una per poter dare avvio alla procedura.

11. `MainFrameKeyEventDispatcher`: questa classe viene inserita tra i listener anche se non è tecnicamente uno di loro; ciò nondimeno però è un oggetto utilizzato dal Controller per stare in ascolto di una azione dell'utente e quindi la sua trattazione è pertinente a questa sezione. Questa classe, invece di essere in ascolto di un evento su una componente grafica, registra le battute sulla tastiera da parte dell'utente, in cerca di particolari combinazioni di tasti, dette anche scorciatoie (*shortcuts*) da tastiera. Quando una determinata combinazione di tasti viene premuta (utilizzando i tasto "Control" in combinazione con una parti-

colare lettera e a volte con il tasto "Alt"), la classe la riconosce e chiama la funzione ad essa associata. Nel sistema ad ogni pulsante del menù principale corrisponde una scorciatoia da tastiera, in modo da poter chiamare una data funzione sia tramite mouse che appunto attraverso la tastiera. Ad esempio, alla combinazione CTRL + O corrisponde la funzione "apri", a CTRL + Z la funzione "annulla" e a CTRL + ALT + S la funzione "salva con nome".

3.3.4.2 Le classi Manager

La classe `EventLogManagerController` gestisce le elaborazioni di ogni singolo processo collegato ai componenti della vista.

Per evitare di avere un'unica singola classe contenente il codice per tutte le elaborazioni possibili nel sistema e rendere così difficile la sua manutenzione (tenendo conto che la suddetta classe già contiene numerose classi interne rappresentati i vari listener descritti nella sezione precedente), sono state create tre classi manager:

1. `EditorManager`: si preoccupa di tutte le operazioni relative alla gestione dei documenti e delle modifiche effettuate ai contenuti degli editor aperti.

2. `AnalysisManager`: si preoccupa di tutte le operazione di validazione, analisi e generazione dei log.

3. `InfoManager`: si preoccupa di tutte le funzionalità più generali connesse alla gestione dell'applicazione.

In questo modo ogni manager gestisce un insieme di operazioni ben preciso ed è facile trovare i metodi che implementano un dato comando con una divisione così chiara dei compiti.

La classe `EventLogManagerController` dunque, una volta ricevuto un comando da parte dell'utente attraverso i componenti della vista, ascoltati a loro volta dai listener che il Controller stesso ha impostato su di loro, in base al menù di provenienza del comando, demanda la sua esecuzione ad uno dei tre manager che faranno di fatto le veci del Controller per lo svolgimento di quella data procedura.

Questi manager infatti vengono creati dal Controller al suo avvio e gli viene fornito un riferimento diretto alla View e al Model, gli stessi che ha il Controller (questi riferimenti non sono mostrati nel diagramma, ma verranno presi in dettaglio nella prossima sezione), in modo tale che, una volta che il Controller ha deciso quale dei tre manager demandare per una data operazione, possano agire come se fosse il Controller stesso ad eseguire l'operazione.

Il Controller inoltre fornisce loro alcuni metodi di utilità da utilizzare quando necessario. Il motivo della centralizzazione di questi metodi è ovviamente la maggiore pulizia del codice e la maggiore facilità di manutenzione dello stesso, essendo procedure comuni a tutti i manager.

La classe `EventLogManagerController` e i tre manager costituiscono di fatto un unico blocco che definisce il comportamento del Controller del pattern MVC. Sono divisi per massimizzare il più possibile la modularità del codice e la sua manutenzione.

Riguardo al manager `AnalysisManager`, come si vede nel diagramma

UML in Figura 3.6, c'è da sottolineare che questa classe contiene a sua volta tre classi interne:

1. `AlphabetSequenceGenerator`: questa classe serve per la generazione di alias alfabetici, utilizzati in fase di analisi come nomi simbolici delle righe e colonne della footprint matrix e associati ciascuno ad un valore ben preciso. Permette di generare infatti sequenze lunghe a piacere di valori alfabetici, dove all'inizio si utilizza un solo simbolo (del tipo "a,b,c,d,...,z") e una volta terminate le possibilità si passa a due, tre, quattro simboli e così via fino a quando non si raggiunge il numero di valori da generare prestabilito (ottenendo sequenze come "aa,ab,ac,...,eba,ebb,ebc,...,zaba,zabb,zabc").

2. `SchemaFactoryErrorHandler`: questa classe è un gestore di errori che possono verificarsi durante la validazione di un documento, nel nostro caso un XSD [2]. Durante la validazione del documento, questa classe registra eventuali errori riscontrati e adotta le misure implementate all'interno della stessa per la gestione di tali errori.

3. `ParserContentHandler`: in maniera del tutto simile alla classe precedente, questa classe implementa un gestore di errori che possono verificarsi durante la validazione di un documento. In questo caso, a differenza della precedente che era specifica per la validazione di do-

[2] *XML Schema Definition* (XSD): è un esempio (*instance*) di schema XML scritto in linguaggio XML Schema. Una XSD definisce il tipo di un documento XML in termini di vincoli, ovvero quali elementi ed attributi possono apparire, in quale relazione reciproca, quale tipo di dati può contenere, ed altro. Può essere usata anche con un programma di validazione, al fine di accertare a quale tipo appartiene un determinato documento XML. Fonte Wikipedia.

cumenti XSD, questa classe può essere utilizzata come gestore degli errori per la validazione di qualsiasi documento preso in esame. Nel nostro caso, viene utilizzata sia per validare log di eventi, secondo lo standard XES, sia per validare documenti XHTML, secondo il DTD[3] da essi specificato.

Oltre alle tre classi annidate sopra elencate, il manager `AnalysisManager` utilizza nelle sue elaborazioni una classe esterna, presente sempre nel pacchetto `controller`: `EventLogGenerator`.

Questa classe implementa la generazione di log di eventi a partire da espressioni testuali e sarà oggetto di analisi nel Capitolo 5.

3.3.4.3 Le classi Parser e Serializer

Le ultime due classi presenti nel pacchetto `controller` sono:

1. `CsvXesParser`: classe per la conversione di documenti in formato XES, ovvero log di eventi, in documenti in formato CSV, ovvero una rappresentazione dei log utilizzando lo standard del formato CSV.

2. `XesCsvSerializer`: classe per la conversione di documenti di log in formato CSV, in documenti in formato XES, ovvero riportando i log di eventi alla loro naturale rappresentazione utilizzando lo standard XES.

Queste due classi sono state create con l'intento di essere collegate all'architettura della libreria Java OpenXES, di cui abbiamo parlato nella Sezio-

[3] *Document Type Definition* (DTD): è uno strumento utilizzato dai programmatori il cui scopo è quello di definire le componenti ammesse nella costruzione di un documento XML. Il termine non è utilizzato soltanto per i documenti XML ma anche per tutti i documenti HTML. Fonte Wikipedia

ne 2.2.4.1. Sono state implementate infatti a partire da componenti generici della libreria (rispettivamente la classe `XParser` e l'interfaccia `XSerializer`) che ne definivano la struttura principale, per specializzarne le funzionalità e creare due classi per la conversione di documenti dal formato XES al formato CSV, e viceversa.

Il Controller all'avvio inserisce queste classi nel registro dei *parser* e dei *serializer* della libreria OpenXES, in modo da renderle disponibili a tutto il sistema.

3.3.5 Vista sulle interazioni tra le classi principali del sistema

Dopo aver visto le varie componenti del pattern MVC separatamente (e le relazioni che intercorrono tra le classi all'interno dei loro pacchetti), in questa sezione prenderemo in considerazione le relazioni che collegano queste componenti, mostrando un diagramma UML dove verranno messe in evidenza le classi principali dei vari pacchetti e le relazioni che intercorrono tra di esse.

In Figura 3.7 è possibile vedere questo diagramma UML "unificato".

Come si può notare tutto inizia a partire dalla classe `EventLogManagerStarter`, che crea i tre componenti `EventLogManagerModel`, `EventLogManagerView` ed `EventLogManagerController`, passando i primi due al terzo in modo che possa interagire con loro per gestire l'intero sistema.

A questo punto `EventLogManagerController`, che adesso ha una associazione con il modello (tramite il riferimento `elmModel`) e con la vista (tramite il riferimento `elmView`) per poterli gestire, crea i suoi manager, `AnalysisMa-`

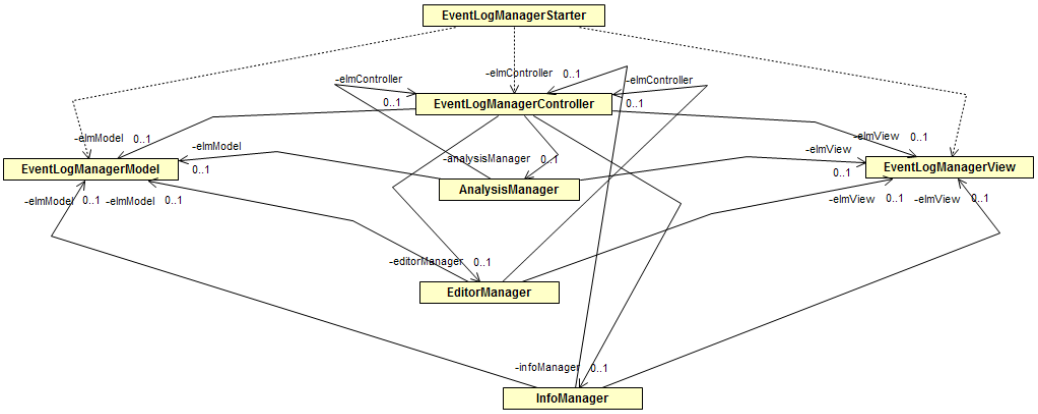

Figura 3.7: Diagramma UML delle principali classi del sistema

`nager`, `EditorManager` e `InfoManager`, mantenendo con loro una associazione tramite i riferimenti riportati nel diagramma per poter demandare loro i compiti da svolgere e passando loro il riferimento alla vista e al modello da gestire.

In questo modo anche i tre manager avranno una associazione con il modello (tramite il riferimento `elmModel`) e con la vista (tramite il riferimento `elmView`), come si può vedere bene nel diagramma UML in Figura 3.7, in modo da poter interagire direttamente con questi componenti senza dover continuamente richiedere alla classe principale di farlo (le interazioni tra le componenti durante una singola elaborazione possono essere davvero molte).

Infine, per completare le associazioni tra queste classi, i tre manager hanno un riferimento alla classe principale del Controller, `EventLogManager-Controller`, in modo da poter interagire con essa per chiamare determinate funzioni messe a disposizione da quest'ultima, per comunicare il risultato delle elaborazioni fatte e non ultimo per interagire con gli altri manager, visto

che è proprio la classe principale a fare da interfaccia di collegamento tra di loro.

Così composto il diagramma delle interazioni tra le componenti principali è completo: in alto la classe del pacchetto `starter` si preoccupa di creare le componenti e dare avvio al sistema; ai lati le classi del pacchetto `model` e di quello `view` gestiscono rispettivamente i dati e l'interfaccia utente, in attesa di comunicazioni da parte del Controller; al centro il pacchetto `controller`, con le sue quattro classi principali che si comportano come un'unica entità comunicando tra loro e con il modello e la vista attraverso i riferimenti descritti, si preoccupa della gestione delle elaborazioni e delle interazioni tra le componenti del sistema.

3.3.6 Le directory del progetto

In questa sezione finale del capitolo si vuole dare una visione su come la struttura del progetto Java, in termini di directory e loro utilizzo all'interno dell'IDE Eclipse, sia stata organizzata.

Le directory del progetto sono le seguenti, riportate in ordine logico di esposizione, definendo per ciascuna di esse il loro utilizzo all'interno di Eclipse:

1. `src`: è la cartella dove risiedono i file sorgenti del progetto (file Java con estensione .java). Al suo interno i file sono suddivisi in sottocartelle che rispecchiano la struttura dei pacchetti dell'applicazione. E' impostata come "Source Folder" all'interno delle impostazioni del progetto di Eclipse.

2. `res`: è la cartella dove si trovano tutti i file non Java del progetto, necessari però al suo funzionamento. E' il caso dei file immagine per la grafica dell'applicazione, dei file testuali di configurazione per le librerie esterne, fino ai file XSD dello standard XES. Ogni gruppo di risorse è diviso in sotto-cartelle tematiche che identificano per cosa quelle risorse sono preposte. E' impostata come "Source Folder" all'interno delle impostazioni del progetto di Eclipse, poter mettere a disposizione il suo contenuto in maniera globale all'interno dell'applicazione e indipendente dalla piattaforma sottostante.

3. `bin`: in questa cartella Eclipse inserisce i file compilati del progetto (file Class con estensione .class) e tutti gli altri file utilizzati dal progetto che non necessitano di compilazione (e che dunque vengono lasciati nella loro forma originale, si pensi ad esempio a file immagine o file testuali di configurazione). Al suo interno i file sono suddivisi in sotto-cartelle che rispecchiano la struttura delle directory impostate come "Source Folders" all'interno di Eclipse.

4. `lib`: in questa cartella sono presenti le librerie esterne utilizzate per lo sviluppo dell'applicazione che sono state descritte nella Sezione 2.2.4. Per ogni libreria vi è una cartella che riporta il suo nome e la sua versione; al suo interno si trova l'archivio Jar che costituisce la libreria e, ove presenti, la cartella `src` con i sorgenti, la cartella `doc` con la documentazione, la cartella `lib` con eventuali librerie da cui a sua volta questa dipende e il file riportante la licenza di utilizzo della libreria.

5. `doc`: in questa cartella Eclipse inserisce la documentazione generata

a partire dai commenti in formato *JavaDoc*[4] presenti all'interno del codice Java dell'applicazione.

6. `uml`: è la cartella dove risiedono i file dei diagrammi UML (.ucls) utilizzati in questo capitolo e la relativa versione in formato immagine (.png), generati automaticamente e tenuti aggiornati ad ogni modifica del codice utilizzando come già accennato il plugin ObjectAid UML Explorer di Eclipse.

[4] *JavaDoc* è un applicativo incluso nel *Java Development Kit* ed utilizzato per la generazione automatica della documentazione del codice sorgente scritto in linguaggio Java. Fonte Wikipedia.

Capitolo 4

SVILUPPO: LA GESTIONE DEI DOCUMENTI

Dopo aver descritto nel Capitolo 3 il procedimento seguito per la progettazione dell'applicazione, possiamo adesso vedere le funzioni che sono state sviluppate all'interno del sistema.

Visto il grande numero di funzioni sviluppate, questa parte verrà suddivisa in tre capitoli, per meglio organizzare l'esposizione delle varie tematiche da trattare.

In questo capitolo ci concentreremo sulle funzioni dell'applicazione per la gestione ed editing dei documenti.

Per i dettagli sul codice e su come le funzioni del sistema siano state sviluppate, rimandiamo il lettore alla consultazione del codice Java messo a disposizione insieme a questo documento (il quale è interamente e dettagliatamente commentato per fornire una spiegazione precisa di ogni implementa-

zione) e alla esaustiva documentazione ad esso allegata (prodotta utilizzando JavaDoc, in modo da generare una documentazione navigabile, nello stesso formato della documentazione delle API Java.

4.1 L'interfaccia dell'applicazione

In questa sezione preliminare, prima di introdurre le funzionalità dell'applicazione, verrà mostrata l'interfaccia utente implementata tramite il pacchetto della vista, in modo da dare la possibilità al lettore di avere chiara la sua struttura durante le successive sezioni.

Per quanto riguarda le componenti dinamiche dell'interfaccia, quelle che vengono create ed eliminate a seconda delle operazioni durante la sessione, ne daremo anche una visione più dettagliata nelle aree tematiche alle quali queste appartengono, in modo da inserirle in un contesto descrittivo appropriato.

In Figura 4.1 viene mostrata una prima visione dell'interfaccia utente per l'applicazione sviluppata, creata attraverso le classi del pacchetto `view` descritto nella Sezione 3.3.3.

In questa prima immagine dell'interfaccia è stata selezionata la vista del primo dei tre menù disponibili: *File*.

I menù infatti in questa applicazione sono tre, selezionabili attraverso i tab in alto. Gli altri due sono *Discovery* e *Info*, mostrati rispettivamente in Figura 4.2 e in Figura 4.3.

In Figura 4.1 il menù File contiene i pulsanti per la gestione dei documenti, dalla loro generazione alla apertura, dalla chiusura al salvataggio, dalla

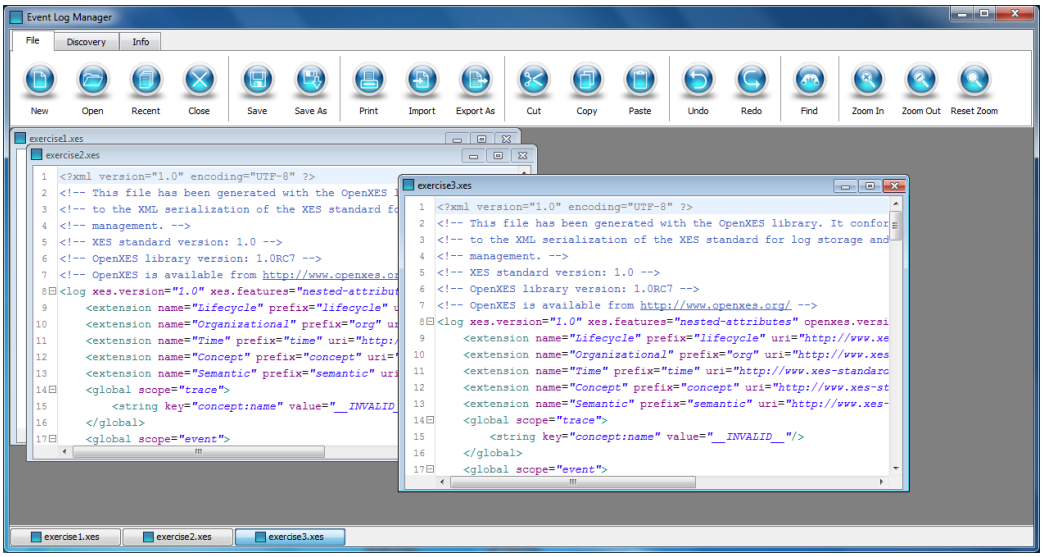

Figura 4.1: L'interfaccia utente, con vista sul menù *File*

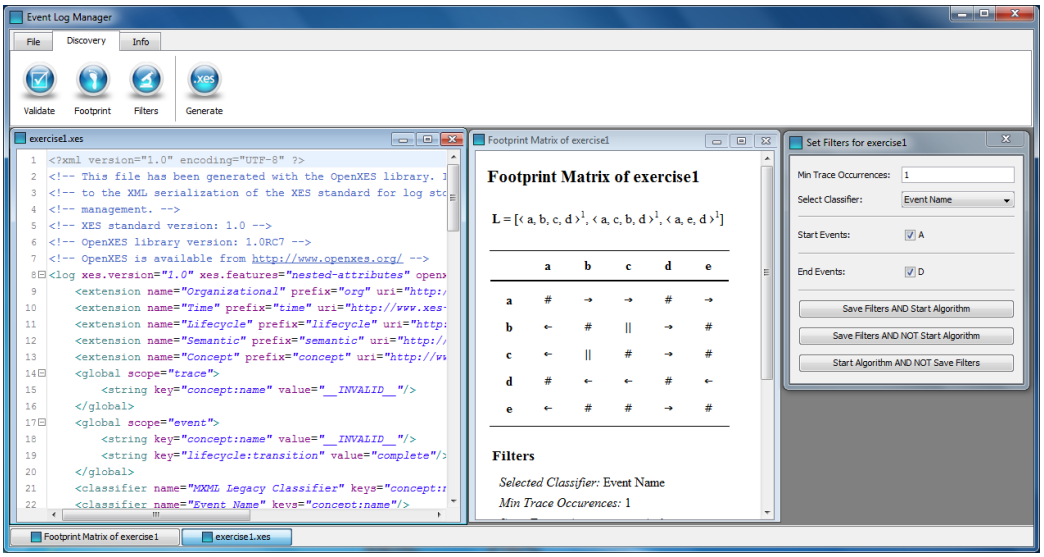

Figura 4.2: L'interfaccia utente, con vista sul menù *Discovery*

stampa all'import ed export, fino ai comandi di aiuto per l'editing del contenuto (è il caso dei pulsanti Cut, Copy e Paste, Undo e Redo e quelli per la gestione dello Zoom).

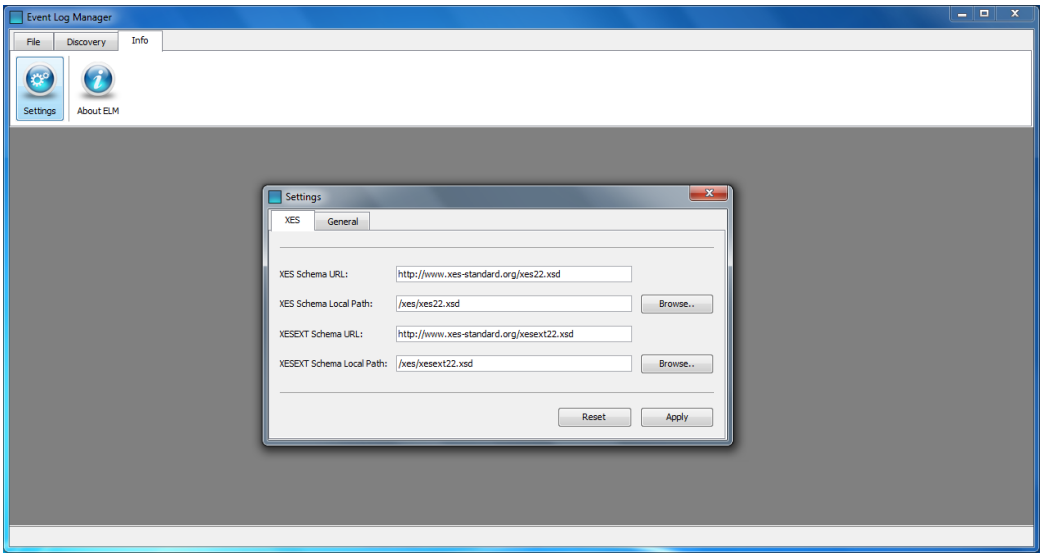

Figura 4.3: L'interfaccia utente, con vista sul menù *Info*

Questi pulsanti sono implementati dalla classe `MenuAreaButton`, che fornisce la descrizione della loro struttura e funzionamento alla selezione da parte dell'utente:

```
1  private void buttonSettings(String toolTip, String actionCommand) {
2     // set dimension
3     this.setPreferredSize(new Dimension(60,80));
4     //set tool tip
5     this.setToolTipText(toolTip);
6     // set name
7     this.setName(actionCommand);
8     //set action command
9     this.setActionCommand(actionCommand);
10    //set position of the text
11    this.setVerticalTextPosition(SwingConstants.BOTTOM);
12    this.setHorizontalTextPosition(SwingConstants.CENTER);
13    //set appearance
14    this.setBorder(new EmptyBorder(new Insets(0,0,0,0)));
15    this.setFocusPainted(false);
16    this.setOpaque(false);
```

```
17    this.setContentAreaFilled(false);
18    this.getModel().addChangeListener(new ChangeListener() {
19       @Override
20       public void stateChanged(ChangeEvent e) { // when the state of this button change
               (hover or not)
21          ButtonModel model = (ButtonModel) e.getSource();
22          if (model.isRollover()) {
23             // set the graphics for the hover state
24             setContentAreaFilled(true);
25          } else {
26             // remove any graphics around the button
27             setContentAreaFilled(false);
28          }
29       }
30    });
31 }
```

Nelle prossime sezioni andremo a vedere il comportamento associato a questi pulsanti e a quelli presenti negli altri menu; per ora vogliamo limitarci a mostrare l'interfaccia sviluppata, per indicare la sua struttura in modo da ottenere una più facile spiegazione in seguito delle funzioni dell'applicazione.

Per rendere la Figura 4.1 più interessante, abbiamo aperto alcuni editor all'interno del sistema. Ci occuperemo di questi editor nella Sezione 4.2, quando parleremo della classe che li implementa: `EventLogEditor`.

Infine, in fondo alla finestra dell'applicazione, sono visibili alcuni pulsanti che riportano i nomi delle finestre interne aperte: questi pulsanti sono i collegamenti per la gestione delle finestre interne aperte, ovvero permettono la loro selezione, minimizzazione, massimizzazione e chiusura[1], in modo da avere un controllo ottimale delle finestre anche quando il loro numero aumenta

[1] A differenza degli altri comandi che sono associati al click sinistro del mouse sul bottone, la chiusura viene attivata tramite il click con il tasto destro del mouse sul bottone e successiva selezione della voce "Close"

in maniera considerevole all'interno dello spazio disponibile e queste tendono inevitabilmente a sovrapporsi.

La classe che implementa questi pulsanti è `FooterAreaButton`, che ne definisce il nome da visualizzare, che sarà identico a quello della finestra associata, la dimensione iniziale ottimale per la visualizzazione per intero del nome (se lo spazio disponibile nella barra lo permette, altrimenti il layout manager `OneRowFlowLayout` di cui parleremo nella Sezione 6.1 si occuperà di gestire questa dimensione in base allo spazio disponibile), il pop-up associato per la gestione della chiusura delle finestra e la connessione con quest'ultima, attraverso un identificatore univoco in comune:

```
1 public FooterAreaButton(String text, Icon icon, boolean selected, int internalFrameID)
      {
2    super(text, icon, selected);
3    buttonSettings(text, internalFrameID);
4 }
5
6 private void buttonSettings(String text, int internalFrameID) {
7    //set name using the id of the internal frame to manage
8    this.setName(String.valueOf(internalFrameID));
9    // set size
10   this.setPreferredSize(new Dimension(getPreferredSize().width + 20, getPreferredSize
         ().height));
11   this.setMinimumSize(this.getPreferredSize());
12   this.setMaximumSize(this.getPreferredSize());
13   // set tool tip
14   this.setToolTipText(text);
15
16   // create a pop-up menu to associate to a right click on this button
17   JPopupMenu rightClickPopup = new JPopupMenu();
18   // create a JMenuItem item with text and an icon if the icon is available, with only
          text otherwise
19   JMenuItem closeItem;
```

```
20    String resourcePath = "/images/close-little.png";
21    URL iconURLClose = this.getClass().getResource(resourcePath);
22    if (iconURLClose != null) {
23       closeItem = new JMenuItem("Close", new ImageIcon(Toolkit.getDefaultToolkit().
              getImage(iconURLClose)));
24    } else {
25       closeItem = new JMenuItem("Close");
26    }
27    // set the action command of the item
28    closeItem.setActionCommand("close-" + String.valueOf(internalFrameID));
29    // set the tool tip of the item
30    closeItem.setToolTipText("Close");
31    // add the item to the pop-up menu
32    rightClickPopup.add(closeItem);
33    // set rightClickPopup as the default pop-up of the button
34    this.setComponentPopupMenu(rightClickPopup);
35 }
```

Per quanto riguarda la Figura 4.2 e la Figura 4.3, la parte alta del menù e quella in basso dei pulsanti per la gestione delle finestre rimangono invariate: funzionano nello stesso modo di quanto descritto per la Figura 4.1, cambiano solo i pulsanti mostrati nel menù in alto e la funzione ad essi associata, che verrà opportunamente descritta nelle successive sezioni.

Quello che cambia è nella parte centrale delle due figure: vengono infatti mostrate adesso una serie di finestre differenti rispetto alla precedente. Il motivo è riassunto nel cercare di dare una prima visione delle varie finestre del sistema, prima di addentrarci nella loro spiegazione. I menù infatti sono indipendenti dalle finestre sottostanti e possono essere cambiati passando da un tab ad un altro, mantenendo inalterate le finestre aperte e massimizzate all'interno della parte centrale. Nel nostro esempio sono diverse solo per poter mostrare differenti finestre insieme ai pulsanti del menù che le generano.

Nella Figura 4.2 sono mostrate per la precisione due finestre interne, a sinistra, e un pannello mobile, a destra: la finestra interna superiore altro non è che un oggetto della classe `FootprintMatrixInfo`, descritta nella Sezione 3.3.3, che implementa una finestra interna per la visualizzazione di codice HTML (utilizzato dall'applicazione per mostrare i risultati delle analisi sui log di eventi); il pannello mobile a destra invece è un pannello del sistema per la visualizzazione dei filtri impostati per l'editor selezionato e per permettere la loro modifica, salvataggio ed eventuale utilizzo per una immediata analisi del log, implementato graficamente dalla classe `FilterPanel`.

Nella Figura 4.3 invece viene visualizzato il pannello delle impostazioni dell'applicazione, dove è possibile modificare ad esempio alcune funzioni dell'editor dei documenti, oppure riferimenti online e locali dei file di definizione dello standard XES e dei documenti HTML.

Anche di queste classi parleremo più in dettaglio in seguito, quando sarà il momento di vedere rispettivamente le funzioni di analisi dei documenti e di gestione dell'applicazione.

4.2 Gestione dei documenti

Dopo la panoramica dell'interfaccia messa a disposizione dell'utente per interagire con il sistema, possiamo adesso introdurre in questa sezione quali sono le funzioni disponibili, all'interno di tale interfaccia, adibite alla gestione dei documenti e quale sia il loro comportamento una volta utilizzate.

Per poter descrivere come il sistema gestisca i vari documenti nei formati disponibili, è necessario prima introdurre e descrivere la classe `EventLogE-`

`ditor`, intorno alla quale ruota l'intera gestione dei documenti da parte delle funzioni dell'applicazione.

La descrizione di questa classe sarà fondamentale anche per le sezioni successive, nelle quali analizzeremo altri processi del sistema.

Successivamente, verranno descritte le varie funzioni di gestione dei documenti all'interno del sistema a partire dai menù visti nella sezione precedente.

4.2.1 La classe EventLogEditor

Come accennato nella Sezione 3.3.3, questa classe definisce la struttura di una particolare finestra interna del sistema: è una estensione della classe Java `JInternalFrame`, classe che a sua volta definisce generiche finestre interne di una GUI sviluppata a partire dalla classe Java `JFrame` (che nel nostro caso è la classe `EventLogManagerVIew`, che appunto estende la classe `JFrame` per definire la finestra principale dell'applicazione). Estendendo questa classe si è potuto definire il contenuto di questa finestra interna e le sue proprietà.

Un esempio di finestra di editor è mostrata in Figura 4.1.

La finestra contiene innanzitutto un editor di testo, per il caricamento e la modifica di documenti in formato testuale, implementato utilizzando la libreria esterna `RSyntaxTextArea` decritta nella Sezione 2.2.4.2. Questa libreria, come ampiamente descritto nella sezione citata, fornisce un editor completo per la gestione e modifica di molti linguaggi di programmazione e mette a disposizione molte funzionalità, come l'evidenziamento della sintassi del codice tramite colorazione (*syntax highlighting*) con la possibilità di scegliere un tema personalizzato per lo stile del font e dei colori del codice, il raggrup-

pamento del codice (*code folding*), il bilanciamento delle parentesi (*bracket matching*), l'indentazione automatica (*auto-indentation*), l'evidenziamento di tutte le occorrenze di un identificatore, variabile o funzione presente nella posizione del cursore, numeri di linea a lato dell'editor, evidenziamento di linea, illimitata registrazione delle operazioni di "Annulla" (*Undo*) e "Ripeti" (*redo*) attraverso l'utilizzo congiunto della classe `RUndoManager` (classe appartenente a questa libreria che estende e migliora il funzionamento di quella delle API Java `UndoManager`) e non ultima una gestione ottimizzata della lettura e scrittura di uno streaming di dati (rappresentanti un documento testuale letto o scritto da/su un file) in modo da trasformare correttamente i vari delimitatori di linea e riga, gli spazi e le tabulazioni da una sorgente ad un'altra senza perdita di informazioni.

La libreria `RSyntaxTextArea` fornisce anche altre funzioni che descriveremo però nelle successive sezioni per mantenere la divisione tematica scelta per questa parte del documento. Per ora ci siamo limitati a tutte le funzionalità messe a disposizione e attivate per la gestione dei documenti di testo.

Quello che è stato necessario è stato semplicemente l'inserimento nella classe `EventLogEditor` dell'editor fornito da questa libreria, ovvero un oggetto della classe `RSyntaxTextArea`, estensione della classe `RTextArea` che a sua volta estende quella Java `JTextArea` (per una totale integrazione e compatibilità con le API Java), e l'attivazione su di esso delle proprietà desiderate:

```
1 // set the text area properties
2 textArea.getDocument().putProperty("owner", this);
3 // use XML syntax highlighting
4 textArea.setSyntaxEditingStyle(SyntaxConstants.SYNTAX_STYLE_XML);
```

```
// set code folding
textArea.setCodeFoldingEnabled(true);
// set antialiasing
textArea.setAntiAliasingEnabled(true);
// set auto indentation
textArea.setAutoIndentEnabled(true);
// set auto close mark-up tags
textArea.setCloseMarkupTags(true);
// set no line wrap
textArea.setLineWrap(false);
// set tab pixel dimension
textArea.setTabSize(4);
// get the selected theme file and apply it to the text area
try {
   String themePath = "/themes/eclipse.xml";
   Theme theme = Theme.load(getClass().getResourceAsStream(themePath));
   theme.apply(textArea);
} catch (IOException e) {
   // no need to do anything, the editor will automatically have the default theme
}
// change the theme font and font size
textArea.setFont(new Font("Courier", Font.PLAIN, defaultFontSize));
```

Quelle mostrate sono le impostazioni personalizzate per il nostro sistema, per le quali abbiamo cambiato il valore di default. Quelle non elencate nel codice sono funzionalità della classe `RSyntaxTextArea` per le quali il valore di default era adeguato ai nostri scopi e quindi non è stato modificato. Mostreremo nelle successive sezioni altre proprietà attivate sul questo editor.

La classe `EventLogEditor` fornisce dunque all'utente un ambiente di gestione e scrittura codice ottimale, ma c'è molto sotto quello che vede l'utente che questa classe fornisce all'intero sistema per una perfetta interconnessione tra le sue varie funzionalità.

Infatti la classe implementa anche un sistema di memorizzazione dello

stato dell'editor e del suo contenuto, in modo da poter informare le altre componenti dell'applicazione in qualsiasi momento queste desiderino dello stato attuale in cui si trova.

Vediamo ora di seguito quali sono le informazioni registrate dalla classe `EventLogEditor` per quanto riguarda la gestione dei documenti:

1. Questa classe permette di registrare in una variabile booleana lo stato "modificato" del contenuto dell'editor, indicando con il valore `true` quando il documento è stato modificato dopo l'ultima salvataggio dello stesso in un file esterno all'applicazione, con `false` quando questo è la rappresentazione di una versione identica presente su un file esterno.

2. Questa classe permette di registrare in una variabile booleana lo stato di "validato" del contenuto dell'editor, indicando con il valore `true` quando il documento è stato controllato e correttamente validato secondo le specifiche dello standard XES (codificate nel file XSD dello standard, utilizzato per la suddetta validazione), con `false` quando, rispetto all'ultima validazione effettuata, il documento è stato modificato e quindi deve essere controllato di nuovo (questo accade anche quando il documento aperto non è ancora mai stato validato e quindi risulta da controllare).

3. Questa classe ricorda la sorgente esterna (il file) dal quale il suo contenuto è stato caricato, o la destinazione sulla quale viene effettuato il salvataggio dello stesso e sul quale fare i salvataggi successivi (è ovvio che le due cose coincidono ad un certo punto della sessione, la sorgente iniziale è la destinazione dei salvataggi successivi, così come la destina-

zione del primo salvataggio diventa a sua volta anche la sorgente del documento). Grazie ad una variabile testuale memorizza il percorso nel file system del file associato a questo oggetto, in modo da poter fornire costantemente un riferimento a quest'ultimo.

Vedremo nelle successive sezioni le altre informazioni registrate dall'editor, nel momento in cui andremo a trattare l'analisi del suo contenuto.

Per ciascuna di queste informazioni, la classe mette a disposizione dei metodi per la lettura (metodi *get*) e per l'aggiornamento (metodi *set*), in modo che le altre componenti che interagiscono con essa possano leggere gli stati di loro interesse ed eventualmente modificarli, a seguito delle loro operazioni.

La classe inoltre permette l'installazione di un listener sull'editor: un oggetto della classe `EventLogEditorDocumentListener`, implementazione della classe `DocumentListener` descritta nella Sezione 3.3.4.1, che resta in ascolto sul contenuto dell'editor e modifica lo stato di quest'ultimo in base alle azioni dell'utente all'interno di esso.

Questo listener alla prima modifica del documento da parte dell'utente, imposta lo stato dell'editor a modificato, sia a livello dell'impostazione vista in precedenza, sia a livello grafico: inserisce infatti accanto al nome della finestra un asterisco, per indicare che il documento è stato modificato rispetto alla versione salvata e necessita di essere salvato di nuovo se non si vuole perdere le modifiche effettuate. Oltre a questa modifica, il listener imposta il contenuto dell'editor come non validato, in quanto modificato rispetto all'ultima modifica e quindi da controllare nuovamente per accertarne la validità. Infine, il listener elimina alcuni riferimenti dell'editor ad oggetti ad esso col-

legati, quali report di analisi, filtri, pannelli aperti, e così via. Vedremo più nel dettaglio questa parte nella sezione dedicata all'analisi dei log.

Dopo aver descritto questa classe e le funzionalità messe a disposizione del sistema, nelle successive sezioni passeremo a vedere come queste siano importanti per la gestione dei documenti e come si integrino con l'interfaccia grafica dell'applicazione.

4.2.2 Apertura dei documenti: Open e Recent

Con riferimento al menù File in Figura 4.1, la prima funzione che andremo a presentare è quella associata al pulsante "Open".

Premendo su questo pulsante (o utilizzando la scorciatoia da tastiera CTRL+O) apparirà la classica schermata di selezione file. In Figura 4.4 ne viene mostrato un esempio.

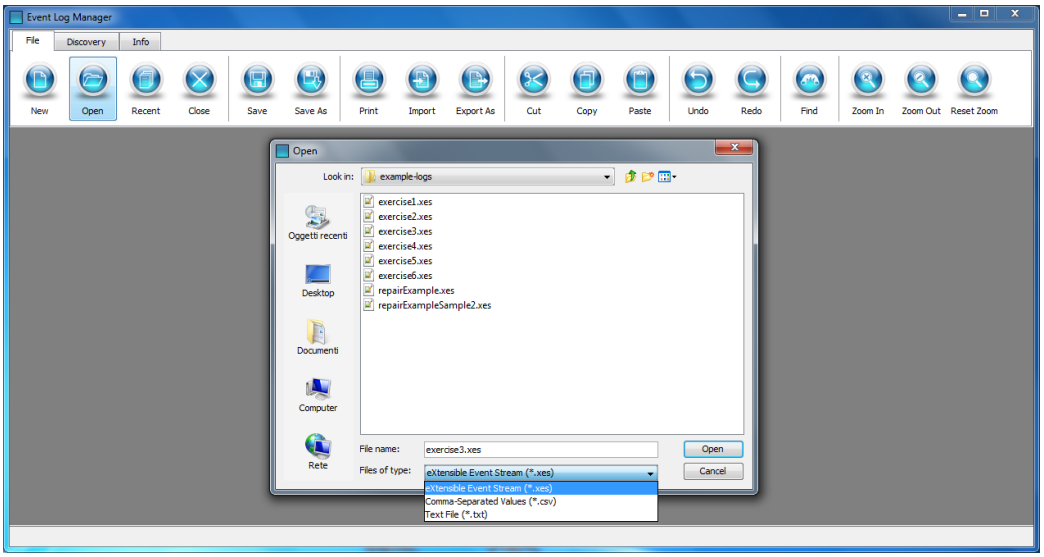

Figura 4.4: La funzione *Open*

Attraverso questa schermata sarà possibile navigare attraverso il file system locale e selezionare il file da aprire.

Vengono forniti tre possibili tipi di file tra cui selezionare:

1. eXtensible Event Stream: questo tipo di file è quello definito per lo standard XES (la cui estensione è ".xes"). É il tipo di file principale che l'applicazione si prefigge di gestire.

2. Comma-Separated Values: questo tipo di file identifica i file CSV (la cui estensione è ".csv"). I file CSV sono file testuali il cui contenuto è scritto secondo le specifiche di questo standard. Nel sistema i file CSV vengono utilizzati come formato alternativo per la definizione di log di eventi, quindi come possibile formato di export per documenti XES, come eventuale import traducibile in un documento XES e per poter scrivere e generare facilmente log di eventi in formato XES. Parleremo in dettaglio dei file CSV nella Sezione 5.1.

3. Plain Text: questo tipo di file identifica i file testuali semplici (la cui estensione è ".txt"). Visto che il sistema si prestava già alla gestione di tipi di file testuali semplici (ovvero senza formattazione, i documenti XES e CSV ne sono un esempio), è stata data la possibilità di scrivere e aprire nell'applicazione qualsiasi file di questo tipo, in modo da poter scrivere o editare delle note ad esempio, oppure aprire e salvare log di eventi o documenti CSV su file di tipo TXT invece che nei formati più standard visti sopra.

Per ogni tipo scelto, la finestra mostra i file con quella estensione, in modo sia da rendere più facile la loro individuazione, sia per evitare che l'utente selezioni file di tipo differente.

Il percorso del file selezionato a questo punto viene controllato, per verificare che il file esista e sia effettivamente utilizzabile dal sistema. In caso negativo, l'applicazione comunica con un messaggio questa situazione di errore e propone all'utente di selezionare un altro file da aprire.

Se il file selezionato risulta già aperto in un editor, il sistema porta quest'ultimo in primo piano e poi informa l'utente con un messaggio, chiedendo di scegliere tra l'opzione di ricaricare il contenuto dell'editor presente nel sistema al quale il file scelto è associato, aprire un nuovo editor con all'interno il contenuto del file selezionato ma mantenendo l'editor già aperto con il contenuto presente e con l'associazione al file selezionato, oppure non fare nessuna azione e prendere come valido l'editor già aperto. In Figura 4.5 è mostrata la scelta che l'applicazione propone all'utente nel caso si verifichi questa eventualità.

Una volta che il procedimento di selezione del file è completato, la funzione si preoccupa di leggere il contenuto del file ed inserirlo in un nuovo editor (che ricordiamo è un oggetto della classe `EventLogEditor`):

```
1 // try-with-resources Statement (close the resource automatically at the end in any
      case)
2 try (InputStream fileStream = new FileInputStream(selectedSourceFile); BufferedReader
      fileReader = new BufferedReader(new InputStreamReader(fileStream))) {
3    // if there is not a target editor to reload, it means we are opening a new file
4    if (targetEventLogEditor == null) {
5       // create a new event log editor set with the target file name, the target file
            content, the absolute path of the target file and its content as not
            modified
```

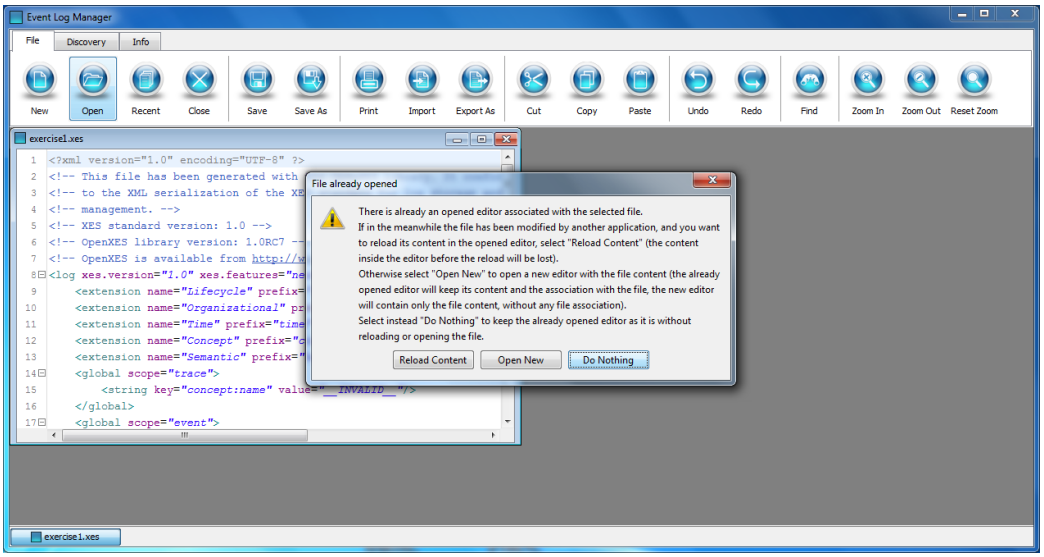

Figura 4.5: La funzione *Open* nel caso di file già aperto

```
6       elmController.newEventLogEditor(targetFile.getFileName().toString(), fileReader,
            selectedSourceFile, false);
7    } else { // if we have a target editor
8       if (openContentOnly) { // if we have to open a new editor with just the content
            in it
9          // create a new event log editor set with default name, the target file
               content, no absolute path and its content as modified
10         elmController.newEventLogEditor(null, fileReader, null, true);
11      } else { // if we have to reload the file instead
12         // set the new content
13         targetEventLogEditor.setTextAreaContent(fileReader);
14         // set the editor content as not modified (because now the editor has the same
               content of the associated file, so it is saved)
15         if (targetEventLogEditor.isContentModified()) {
16            // if not already set as false, set it
17            targetEventLogEditor.setContentModified(false);
18            // remove the asterisk (if the content is modified, the title has an
                  asterisk at the end)
19            String savedTitle = targetEventLogEditor.getTitle();
20            savedTitle = savedTitle.substring(0, savedTitle.length()-1);
```

```
21            elmController.modifyJInternalFrameTitles(targetEventLogEditor, savedTitle);
22          }
23          // set the editor content as not validated (because it is a new content)
24          targetEventLogEditor.setContentValidated(false);
25          // the title and the associated file are already set, then no needs to modify
                them
26        }
27    }
28    // add the path of the opened file to the recent list in the Model
29    elmModel.addRecentLog(selectedSourceFile);
30 } catch(IOException e) {
31    elmView.showErrorMessage("Unable to read the file. Please try again.");
32 }
```

Il procedimento di lettura è molto semplice, grazie alle funzionalità messe a disposizione dalla classe `EventLogEditor`, poiché è sufficiente creare un nuovo oggetto di questa classe passando come parametri il nome della finestra, l'oggetto Java aperto sullo streaming dei dati del file, il percorso del file selezionato sul file system (se esiste, in caso di file già aperto e scelta di aprire un nuovo editor, quest'ultimo non avrà alcun file associato) e se l'editor debba considerare il contento come modificato oppure no (come prima, un editor aperto su un file considererà il contenuto come non modificato rispetto al file, in quanto appena letto, invece un editor a cui viene passato del testo da visualizzare ma senza file associato, lo considererà come modificato rispetto alla sorgente, in quanto questa non esiste).

Una volta inserita la finestra di editing nel corpo centrale dell'applicazione, viene creato un pulsante di tipo `FooterAreaButton` ed inserito nella barra dei pulsanti in fondo all'applicazione: questo oggetto servirà da collegamento con la finestra creata per poterla selezionare, minimizzare, massimizzare

e chiudere, come descritto nella Sezione 4.1. Vedremo queste funzionalità in dettaglio nella Sezione 6.3.

Come si può notare del codice appena mostrato, alla fine del procedimento di apertura, il percorso del file aperto viene salvato in una struttura dati del modello, una lista a capacità limitata di tipo LIFO implementata dalla classe `MostRecentElementList<E>`. Questa lista memorizza i percorsi degli ultimi documenti aperti (un solo elemento nella lista per documento, se un documento viene riaperto ed era già nella lista, il suo percorso viene spostato in cima come più recente), in ordine dal più recente al meno recente, in numero limitato (di default il limite è uguale a dieci, ma può essere modificato nelle impostazioni dell'applicazione). Quando la lista è piena ed un nuovo elemento viene inserito in essa, il meno recente viene eliminato per fargli posto.

Quando un utente preme sul pulsante "Recent", quello che apparirà sarà un sotto-menù contenente proprio gli elementi di questa lista, in ordine. Premere su uno di essi equivale a scegliere di riaprire quel documento all'interno del sistema, e questo verrà fatto utilizzando la funzione Open precedentemente mostrata: in questo caso la funzione prende direttamente il percorso del file da aprire (e quindi non apre nessuna finestra di scelta), dopodiché effettua gli stessi controlli (file esistente e documento già aperto o meno) e le stesse scelte in base alle risposte dell'utente alle sue richieste. La chiamata della funzione Open fa anche si che il documento appena aperto venga reinserito all'interno della lista dei documenti recenti, spostando di fatto quest'ultimo in cima alla lista stessa rispetto alla posizione in cui si trovava precedentemente.

4.2.3 Salvataggio dei documenti: Save e Save As

Dopo aver aperto un documento ed averlo modificato, quello che possiamo fare è salvarlo.

L'interfaccia mette a disposizione i due pulsanti *Save* e *Save As*, che si comportano nel classico modo che possiamo riscontrare in tutte le applicazioni.

Come abbiamo detto nella sezione precedente, quando si effettua una modifica ad un documento appena aperto, o comunque salvato su un file esterno, questo assume lo stato di modificato, con relativa segnalazione all'utente tramite asterisco alla fine del nome della finestra.

Quando ad un documento che presenta lo stato di modificato, si applica la funzione Save utilizzando l'apposito pulsante (o la scorciatoia da tastiera CTRL+S), quello che accade è semplice: la funzione controlla se il documento ha già un file associato (nel qual caso è un documento aperto da un file esterno oppure precedentemente è stato già salvato su un file); se lo ha, procede al salvataggio del contenuto del documento all'interno del file associato (che se non esiste più poiché cancellato nel frattempo, viene ricreato prima di procedere al salvataggio), rimuovendo infine l'asterisco dal nome e impostando l'editor come non modificato.

Se durante questa procedura, la funzione riscontra l'assenza di un file associato (è il classico caso di un quando si apre un editor vuoto e vi si scrive del testo, il documento risulta modificato ma mai salvato, e quindi non ha ancora un file di destinazione sul quale essere salvato) oppure il file indicato non esiste più sul file system (ovvero è stato cancellato nel frattempo oppure

l'applicazione non può accedervi) e la funzione non riesce a ricrearlo, questa demanda le operazioni di salvataggio alla funzione Save As.

```
1  // if this editor does not have an associated file
2  if ((absolutePath == null)||(absolutePath.equals(""))) {
3     // ask for the saving the "save as" procedure
4     return saveAsEventLog(null);
5  } else {
6     // create a Path object using the String path
7     Path targetFile = Paths.get(absolutePath);
8     // get the target file name
9     String targetFileName = targetFile.getFileName().toString();
10    // if the file still exists
11    if (Files.exists(targetFile)) {
12       // if the editor content has not been modified by the user
13       if (!eventLogEditor.isContentModified()) {
14          // the editor as a valid associated file and no content to save
15          elmView.showInfoMessage("The selected document has already been saved.");
16          fileSaved = true;
17          return fileSaved;
18       }
19    } else if (Files.notExists(targetFile)) { // if the target file does not longer
          exist, we create it again
20       try {
21          Files.createFile(targetFile);
22       } catch (IOException e) {
23          elmView.showErrorMessage("Unable to save the document: error while creating
                the file. \n"
24                                + "Please select another file where to save.");
25          // ask for the saving the "save as" procedure
26          return saveAsEventLog(targetFileName);
27       }
28    } else { //if the process cannot determine if the file exists or not, because it
          does not have access to the file
29       elmView.showErrorMessage("Unable to save the document. \n"
30                             + "The program does not have access to the file associated
                                  with this editor. \n"
31                             + "Please select another file where to save.");
```

```
32      // ask for the saving the "save as" procedure
33      return saveAsEventLog(targetFileName);
34    }
35    // write the content of the event log in the target file
36    try (OutputStream fileStream = new FileOutputStream(targetFile.toString());
          BufferedWriter fileWriter = new BufferedWriter(new OutputStreamWriter(
          fileStream))) {
37      // start writing the file and set the application in a busy state in the
            meanwhile
38      elmView.setBusyState(true);
39      eventLogEditor.getTextAreaContent(fileWriter);
40      elmView.setBusyState(false);
41    } catch (IOException e) {
42      elmView.setBusyState(false);
43      elmView.showErrorMessage("Unable to save the document in the selected file. \n"
44                          + "Please select another file where to save.");
45      // ask for the saving the "save as" procedure
46      return saveAsEventLog(targetFileName);
47    }
48    if (eventLogEditor.isContentModified()) {
49      // set the event log editor as saved and remove the asterisk from its title
50      eventLogEditor.setContentModified(false);
51      String savedTitle = eventLogEditor.getTitle();
52      savedTitle = savedTitle.substring(0, savedTitle.length()-1);
53      elmController.modifyJInternalFrameTitles(eventLogEditor, savedTitle);
54    }
55    ...
56  }
```

Questa seconda funzione di salvataggio, sia se chiamata a partire da una precedente Save, sia se invocata direttamente dal suo pulsante nell'interfaccia (o attraverso la scorciatoia da tastiera CTRL+ALT+S), si comporta nello stesso modo.

Per prima cosa apre una finestra di scelta, dove l'utente può scegliere il file di destinazione sul quale salvare: l'operazione di scelta è identica a

quella spiegata per la funzione di apertura di un documento, in quanto i tipi di file sui quali si può salvare, sono gli stessi che si possono aprire (ovvero XES, CSV e TXT). L'unica differenza risiede nel fatto che, mentre nella funzione di apertura bisogna selezionare un file, qui si può non solo fare questo, ma anche crearne uno nuovo: è sufficiente inserire il nome del nuovo file e selezionare sotto il suo tipo ed il sistema lo creerà per poi utilizzarlo per il salvataggio. É importante notare che la scelta del tipo sovrascrive le eventuali estensioni definite nel campo del nome del file. Ad esempio scrivere "file.xes" come nome del file e poi selezionare come tipo CSV, comporterà la creazione di un file di nome "file.xes.csv". Nel caso però si definisca un nome con estensione e poi si selezioni lo stesso tipo di file, l'estensione aggiunta non sarà doppia ma singola (il sistema si accorge che un file di nome "file.xes" che deve essere salvato come tipo XES, è già pronto per il salvataggio e non occorre aggiungere l'estensione ".xes" alla fine, in quanto già presente).

In Figura 4.6 è mostrato il pannello di selezione del file di destinazione del documento da salvare. In particolare viene inserito un nuovo nome per il file, quindi l'intenzione è quella di crearne uno nuovo dove salvare e non di utilizzarne uno già esistente.

Dopo che il nome del file ed il percorso sono stati selezionati con successo, la funzione controlla se questo file esiste (visto che come detto si possono selezionare file esistenti tanto quanto crearne di nuovi): se non esiste, lo crea e poi vi salva il contenuto dell'editor, se esiste invece, chiede all'utente se desidera o meno sovrascrivere il file, procedendo con il salvataggio in caso di risposta negativa, con la riproposizione della finestra di scelta file nel caso di risposta negativa alla sovrascrittura di quel file (quindi si rende necessario

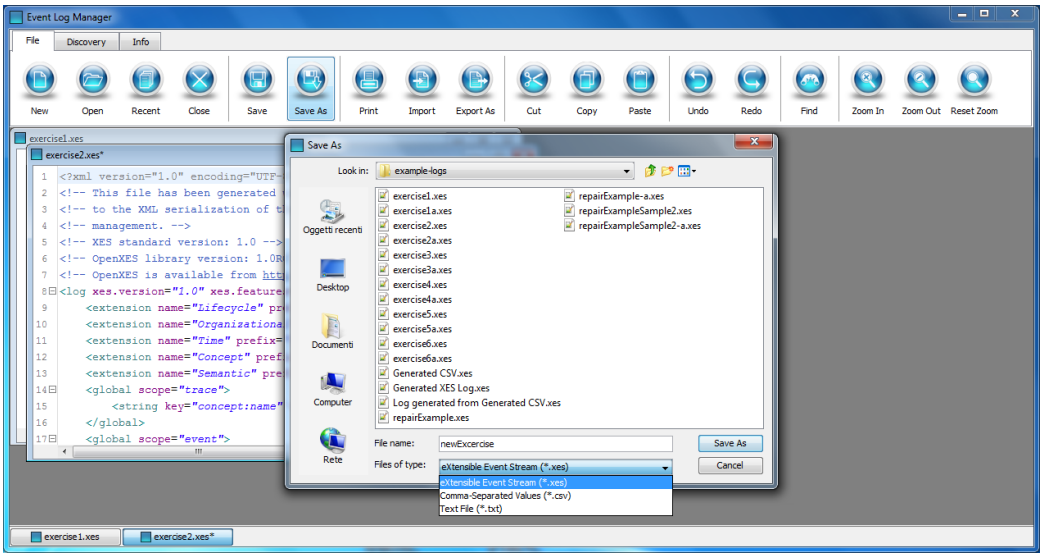

Figura 4.6: La funzione *Save As*

selezionarne un altro).

Al termine delle operazioni di salvataggio, la funzione Save As non solo elimina l'eventuale asterisco alla fine del nome della finestra, ma aggiorna anche il percorso del file associato all'editor con quello utilizzato per il salvataggio (quindi aggiorna se già esistente, o inserisce per la prima volta se il documento non era mai stato salvato prima).

```
1  // open a dialog to make the user select a file where to save, using filters
2  String selectedDestinationFile = elmView.showFileSelectionDialog("Save As",
       selectedFile, xesFilter, csvFilter, txtFilter);
3  // check if the user have selected a file, otherwise do nothing
4  if (selectedDestinationFile != null) {
5    // create a Path object using the String path
6    Path targetFile = Paths.get(selectedDestinationFile);
7    // get the target file name
8    String targetFileName = targetFile.getFileName().toString();
9    // remember if we create a new file
10   boolean newFileCreated = false;
11   // if the file still exists
```

```
12  if (Files.exists(targetFile)) {
13     // retrieve the user's choice
14     String message = "The file already exists. Do you want to overwrite it?";
15     String title = "Save As";
16     Object[] options = {"Yes", " No "};
17     int choice = elmView.showOptionDialog(message, title, JOptionPane.WARNING_MESSAGE
           , options, options[1]);
18     if (choice != 0) { // if the user select No (choice=1) or close the dialog (
           choice=-1)
19        return saveAsEventLog(targetFileName);
20     } // otherwise go on to the saving procedure
21  } else if (Files.notExists(targetFile)) { // if the target file does not exist
22     try {
23        // create it
24        Files.createFile(targetFile);
25        newFileCreated = true;
26     } catch (IOException e) {
27        elmView.showErrorMessage("Unable to save the document: error while creating
              the file. \n"
28                              + "Please select another file where to save.");
29        return saveAsEventLog(targetFileName);
30     }
31  } else { //if the process cannot determine if the file exists or not, because it
       does not have access to the file
32     elmView.showErrorMessage("Unable to save the document. \n"
33                           + "The program does not have access to the selected file. \n"
34                           + "Please select another file where to save.");
35     return saveAsEventLog(targetFileName);
36  }
37  // write the content of the event log in the target file
38  try (OutputStream fileStream = new FileOutputStream(targetFile.toString());
       BufferedWriter fileWriter = new BufferedWriter(new OutputStreamWriter(
       fileStream))) {
39     // start writing the file and set the application in a busy state in the
           meanwhile
40     elmView.setBusyState(true);
41     eventLogEditor.getTextAreaContent(fileWriter);
```

```
42        elmView.setBusyState(false);
43     } catch (IOException e1) {
44        elmView.setBusyState(false);
45        elmView.showErrorMessage("Unable to save the document in the selected file. \n"
46              + "Please select another file where to save.");
47        if (newFileCreated) {
48           try {
49              Files.delete(targetFile);
50           } catch (IOException e2) {
51              ...
52           }
53        }
54        return saveAsEventLog(targetFileName);
55     }
56     // set the event log editor as saved
57     eventLogEditor.setContentModified(false);
58     // if the internal frame has a different title than the file name where we just
           saved it, or it has an asterisk at the end showing that it was not saved, then
           we need to update it
59     if (!eventLogEditor.getTitle().equalsIgnoreCase(targetFileName)) {
60        elmController.modifyJInternalFrameTitles(eventLogEditor, targetFileName);
61     }
62     // get the absolute path of the file associated with this editor, if exists
63     String eventLogEditorAbsolutePath = eventLogEditor.getAbsolutePath();
64     // if the file path where we just saved is different from the last one, we update it
65     if (eventLogEditorAbsolutePath == null) {
66        eventLogEditor.setAbsolutePath(selectedDestinationFile);
67     } else if (!eventLogEditorAbsolutePath.equalsIgnoreCase(selectedDestinationFile)) {
68        eventLogEditor.setAbsolutePath(selectedDestinationFile);
69     }
70     ...
71  }
```

Entrambe queste funzioni sono state implementate esclusivamente per l'editor definito dalla classe `EventLogEditor`, in quanto composto da contenuto prettamente testuale. Per salvare il contenuto di altre finestre aperte

dal sistema, come ad esempio quelle contenenti i report dell'analisi dei log, il sistema suggerisce in automatico l'utilizzo della funzione di *Export*, che descriveremo nella Sezione 5.3.

4.2.4 Chiusura di una finestra: Close

Quando l'utente decide di voler chiudere una finestra di editing, il modo più semplice è utilizzando il pulsante di chiusura sulla barra della finestra, oppure il pulsante *Close* nel menù principale (la cui scorciatoia da tastiera è CTRL+W).

Sia il pulsante che la barra principale della finestra infatti utilizzano lo stesso procedimento, semplicemente partendo da due listener diversi (uno in ascolto sui pulsanti del menù (`MenuAreaButtonListener`), uno in ascolto sulle azione che l'utente compie sulla finestra(`DefaultInternalFrameAdapter`)).

Questa funzione si comporta diversamente a seconda che la finestra da chiudere sia una di editing, oppure sia una `FootprintMatrixInfo` utilizzata per la visualizzazione dei report delle analisi.

```
1  // if the frame is an EventLogEditor
2  if (selectedInternalFrame instanceof EventLogEditor) {
3     ...
4     // if the editor content has not been modified by the user, check the associated
          file and then decide
5     if (!eventLogEditor.isContentModified()) {
6        if ((absolutePath == null)||(absolutePath.equals(""))) {
7           // this is a new empty event log, so we can just close it
8           eventLogEditor.dispose();
9           fileClosed = true;
10       } else {
11          // create a Path object using the String path
12          Path targetFile = Paths.get(absolutePath);
```

```
13          // if the file still exists
14          if (Files.exists(targetFile)) {
15             // the editor has no modifications to save and its associated file still
                    exists, so we can close it
16             eventLogEditor.dispose();
17             fileClosed = true;
18          } else if (Files.notExists(targetFile)) { // if the target file does not
                longer exist
19             // set a specific message for the save procedure and the name for the file
20             ...
21          } else { //if the process cannot determine if the file exists or not, because
                it does not have access to the file
22             // show an error message to the user and then set the saving name for the
                    file
23             ...
24          }
25       }
26    }
27    // if the file is still open, it means we need to save it
28    if(!fileClosed) {
29       // retrieve the user's choice, using the message decided before
30       ...
31       int choice = elmView.showOptionDialog(message, title, JOptionPane.
             QUESTION_MESSAGE, options, options[0]);
32       // check user's decision
33       if (choice == 0) { // if YES, call the save method
34          boolean fileSaved;
35          // if we have to save in a new location
36          if (location != null) {
37             fileSaved = saveAsEventLog(location);
38          } else { // otherwise just save in the same location
39             fileSaved = saveEventLog();
40          }
41          if (fileSaved) { // if we have successfully saved the file, then set the
                editor as closable
42             eventLogEditor.dispose();
43             fileClosed = true;
```

```
44          }// if we have not saved the file, then do not close the editor because its
                content has not been saved
45      } else if (choice == 1) { // if NO, just set the internal frame as closable,
            without saving it
46          eventLogEditor.dispose();
47          fileClosed = true;
48      }
49      // if the choice is Cancel (2) or the user closed the dialog (-1), do nothing
50    }
51  } else if (selectedInternalFrame instanceof FootprintMatrixInfo) { // if the frame is a
        FootprintMatrixInfo
52    ...
53    // check if the file has already been exported
54    if (footprintMatrixInfo.isContentExported()) { // if the document has already been
          exported by the user, check the associated file and then decide
55      // create a Path object using the String path
56      Path targetFile = Paths.get(footprintMatrixInfo.getAbsolutePath());
57      // if the file still exists
58      if (Files.exists(targetFile)) {
59          // it has been exported and its associated file still exists, so we can close
                it
60          footprintMatrixInfo.dispose();
61          fileClosed = true;
62      } else if (Files.notExists(targetFile)) { // if the target file does not longer
            exist
63          // set a specific message for the export procedure and the name for the file
64          ...
65      } else { //if the process cannot determine if the file exists or not, because it
            does not have access to the file
66          // show an error message to the user, then set a specific message for the
                export procedure and the export name for the file
67          ...
68      }
69    }
70    // if the file is still open, it means we need to export it
71    if (!fileClosed) {
72      // retrieve the user's choice, using the message decided before
```

```
73      ...
74      int choice = elmView.showOptionDialog(message, title, JOptionPane.
            QUESTION_MESSAGE, options, options[0]);
75      // check user's decision
76      if (choice == 0) { // if YES, call the export method
77        // if we have to export in a new location or in the same is decided by the
              value of location (null or String)
78        boolean fileExported = exportDocument(location);
79        if (fileExported) { // if we have successfully exported the file, then set the
              editor as closable
80          selectedInternalFrame.dispose();
81          fileClosed = true;
82        }// if we have not exported the file, then do not close the editor because its
              content has not been saved
83      } else if (choice == 1) { // if NO, just set the internal frame as closable,
            without exporting it
84        selectedInternalFrame.dispose();
85        fileClosed = true;
86      }
87      // if the choice is Cancel (2) or the user closed the dialog (-1), do nothing
88    }
89 } else { // for every other type of frame, close them directly
90    selectedInternalFrame.dispose();
91    fileClosed = true;
92 }
```

Nel primo caso, si controlla innanzitutto se il contenuto dell'editor risulta modificato: se non è stato modificato e l'editor ha un file associato esistente e accessibile, o se si tratta di un editor nuovo e vuoto, la finestra viene chiusa; se il documento non è stato modificato ma il file associato risulta cancellato o non accessibile, o se il contenuto dell'editor risulta modificato, la funzione informa l'utente di queste situazioni e chiede se si vuole salvare il documento prima di chiuderlo.

In Figura 4.7 è mostrata questa situazione, dove, come possiamo no-

tare dall'asterisco accanto al nome, il documento "exercise2.xes" è stato modificato e non salvato prima di richiederne la chiusura.

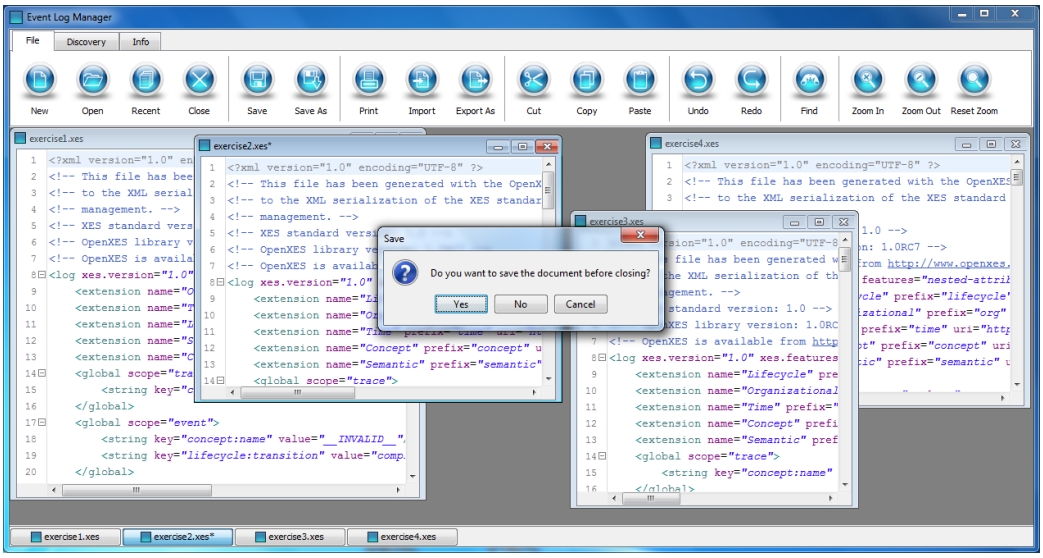

Figura 4.7: La funzione *Close* nel caso di un documento di testo

Oltre alla possibilità di annullare la procedura di chiusura e mantenere il documento aperto, è ovviamente possibile rispondere alla domanda in maniera negativa, ottenendo che l'editor verrà chiuso e il contenuto non salvato, oppure positiva: in questo caso la funzione demanda le operazioni di salvataggio alla procedura Save, nel caso in cui l'editor abbia un file di destinazione impostato, alla procedura Save As, altrimenti.

Al termine delle procedure di salvataggio, se concluse dall'utente in maniera positiva, la finestra viene chiusa. In caso di annullamento della procedura di salvataggio invece, la funzione Close lascia inalterata la finestra all'interno dell'applicazione.

Nel secondo caso, quello in cui la finestra sia una `FootprintMatrixInfo`, la funzione chiede all'utente se si vuole esportare il documento, e non salvare.

Questo deriva dal fatto che, mentre il contenuto di un editor è testuale e può essere salvato nei modi visti nella procedura di salvataggio, il contenuto di questa finestra è in HTML e deve essere trattato diversamente: se infatti si salvasse in maniera testuale il contenuto, otterremmo semplicemente un file di tipo testuale con all'interno codice HTML.

In conseguenza di questo, l'eventuale salvataggio del contenuto viene demandato alla funzione di export, che si occuperà di processare la richiesta in maniera appropriata. Vedremo in dettaglio la funzione di export nella Sezione 5.3, per ora citiamo il fatto che questa funzione permetterà di esportare il contenuto di questa finestra in vari formati che ne permetteranno la corretta visualizzazione. I formati disponibili sono: HTML (apribile in qualsiasi browser ed immediatamente visualizzabile), PDF, PNG e LaTeX (quest'ultimo genera un documento in formato TeX a partire dal codice HTML della finestra, che poi potrà essere a sua volta modificato e compilato in appositi programmi per questo linguaggio).

Il comportamento della funzione Close in questa situazione è comunque pressoché identico alla precedente situazione: se il documento risulta mai esportato in HTML, chiede all'utente se vuole o meno esportare il contenuto, procedendo in caso negativo alla chiusura immediata della finestra, in caso affermativo all'avvio della funzione di export e alla chiusura della finestra solo se il procedimento si è concluso positivamente (in maniera analoga al comportamento tenuto nel caso delle finestre `EventLogEditor`); se il documento risulta esportato almeno una volta in HTML ed il file dove è stato esportato esiste, la finestra viene chiusa, se il file non esiste più o non è accessibile, si procede alla richiesta di export all'utente.

In Figura 4.8 viene mostrato il differente messaggio quando tentiamo di chiudere un report di analisi anziché un editor di testo.

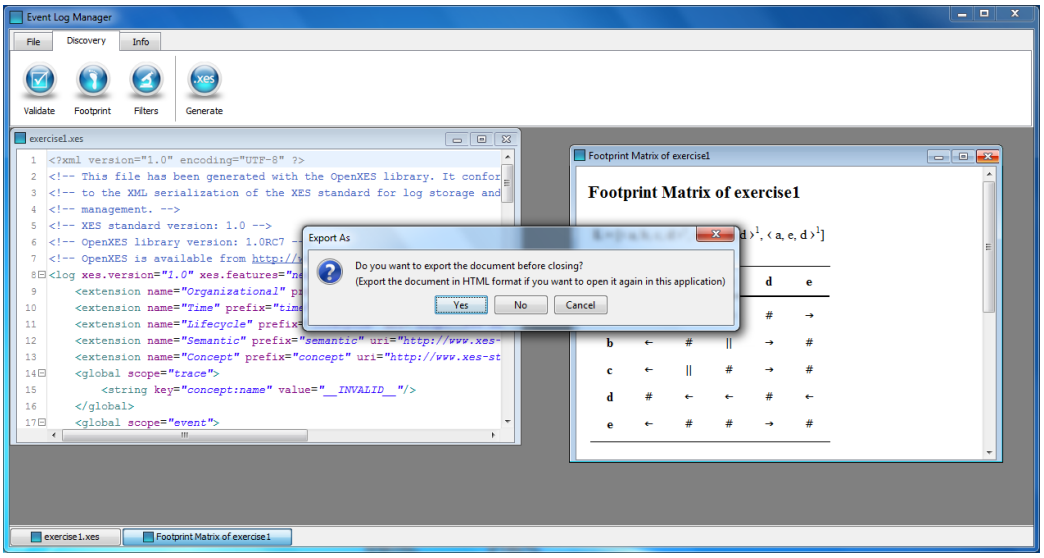

Figura 4.8: La funzione *Close* nel caso di un documento HTML

Abbiamo appena detto che la discrezionalità sulla richiesta di effettuare o meno un export dipende dal fatto che il contenuto della finestra sia stato o meno esportato in HTML. Il motivo per il quale solo questo formato vale come "salvataggio" è semplice: mentre gli altri formati sono non modificabili (PDF e PNG sono documenti finali, non più apribili in editor di testo per essere modificati) oppure scritti in linguaggi diversi dall'originale (una volta generato il documento in LaTeX, questo non può essere usato per tornare alla versione HTML), il formato HTML è l'esatta rappresentazione del contenuto della finestra, che semplicemente mostra la compilazione del suddetto codice all'utente, ma che sotto mantiene il riferimento al codice sorgente del documento. Esportando dunque il documento in HTML si ottiene una versione identica a quella interna all'applicazione: questa versione può allora essere

importata di nuovo nel sistema utilizzando la funzione di *Import* (che vedremo nella Sez. 5.3), in maniera del tutto simile ad una procedura di apertura di un documento testuale all'interno dell'applicazione, come abbiamo visto in precedenza. É proprio questa possibilità di salvataggio esterno e riapertura nel sistema che caratterizza l'export in HTML come un formato valido come definizione di "salvataggio" del documento.

É importante infine notare come il listener in ascolto sulla finestra, `DefaultInternalFrameAdapter`, quando una di queste viene chiusa, si occupi anche dell'eliminazione del relativo pulsante `FooterAreaButton` all'interno della barra in fondo alla finestra principale e dell'eventuale `FilterPanel` aperto ad esso associato, in modo da rimuovere ogni riferimento alla finestra e componente ad essa collegato.

4.2.5 Stampa di un documento: Print

L'applicazione, attraverso il pulsante "Print" dell'interfaccia, permette di stampare i documenti testuali aperti in finestre `EventLogEditor`.

```
1  // create an header and a footer for the document
2  MessageFormat header = new MessageFormat(eventLogEditor.getTitle());
3  MessageFormat footer = new MessageFormat("Page - {0}");
4  // get the actual font size of the editor
5  int fontSize = eventLogEditor.getFontSize();
6  // set the font size to a good value for a good printing of the code
7  eventLogEditor.setFontSize(8);
8  // get the textArea where is the code to print
9  RSyntaxTextArea textArea = eventLogEditor.getTextArea();
10 // set the line wrap to true so the code can wrap inside the document
11 textArea.setLineWrap(true);
12 try {
13    // try to open the print settings dialog
```

```
14    // if the user select OK in the dialog print the document and return true, false
          otherwise
15    isPrinted = textArea.print(header, footer);
16 } catch (PrinterException e) {
17    elmView.showErrorMessage("Unable to print the selected document. Please try again.")
          ;
18 }
19 // restore the font size and the no-wrap setting of the textArea
20 eventLogEditor.setFontSize(fontSize);
21 textArea.setLineWrap(false);
```

In Figura 4.9 viene mostrato il pannello per la stampa del documento che viene aperto dall'applicazione, attraverso il quale è possibile selezionare il tipo di stampante che si desidera utilizzare e impostare le opzioni di stampa prima di procedere.

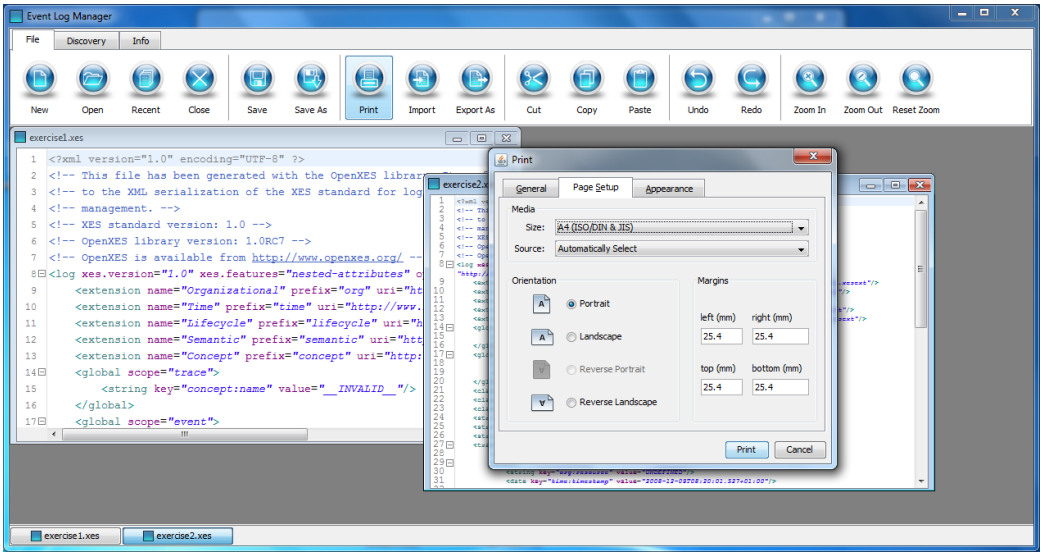

Figura 4.9: Il pannello per la stampa

Per quanto riguarda invece i documenti presenti nelle finestre `FootprintMatrixInfo`, il sistema suggerisce all'utente di utilizzare la funzione Export,

selezionando PDF come tipo di file, per ottenere una resa grafica migliore del documento e successivamente stampare il PDF generato.

4.2.6 Funzioni di utilità per l'editing dei contenuti

Come è stato accennato già più volte, la classe `RSyntaxTextArea` estende quella Java `JTextArea`. Quest'ultima fornisce alcuni metodi di utilità da poter utilizzare all'interno dell'editor, quali taglia, copia ed incolla, che la classe `RSyntaxTextArea` eredita.

Oltre a questi metodi, la libreria `RSyntaxTextArea` estende attraverso la classe `RUndoManager` la classe delle API Java `UndoManager`, migliorandola, per poter fornire le funzioni di annulla e ripeti.

La classe delle API Java infatti ha un problema ormai noto in ambito di sviluppo, ovvero implementa le funzioni di annulla e ripeti su singolo carattere: se scriviamo una frase e proviamo ad utilizzare i metodi `undo` e `redo` della classe `UndoManager`, quello che otteniamo è un annullamento o ripetizione della digitazione carattere per carattere, e non gruppi di parole o operazioni. Inoltre questa classe ha una memoria limitata e non permette di annullare tutte le modifiche fatte, ma solo le ultime.

La classe `RUndoManager` implementa invece un sistema di raggruppamento dei caratteri digitati, in modo da fornire funzioni di annulla e ripeti come quelle che troviamo nei normali editor di testo, nei quali annulliamo l'ultima operazione fatta o l'ultima parola/blocco di parole scritte. Questa classe inoltre ha una memoria "infinita", ovvero si ricorda di tutte le modifiche fatte dall'apertura dell'editor, permettendo così di avere funzioni di annulla

e ripeti che possono tornare indietro fino alla prima digitazione di contenuti nell'area di testo e ritorno.

Queste funzioni di utilità così definite vengono fornite dall'applicazione sia attraverso i pulsanti nel menù principale *Cut*, *Copy*, *Paste*, *Undo* e *Redo* che possiamo vedere in Figura 4.1, sia soprattutto attraverso scorciatoie da tastiera.

Le scorciatoie sono quelle classiche per queste operazioni: CTRL+X per la funzione Cut, CTRL+C per Copy, CTRL+V per Paste, CTRL+Z per Undo e CTRL+Y per Redo.

Queste funzioni non fanno altro che richiamare sull'editor `EventLogEditor` selezionato il rispettivo metodo, che in automatico effettuerà le operazioni di taglia, copia e incolla utilizzando quelle fornite dal sistema operativo sottostante e di annulla e ripeti utilizzando la classe `RUndoManager` messa a disposizione dalla libreria.

```
1  public void cutEventLog() {
2     // take the selected internal frame
3     JInternalFrame selectedInternalFrame = elmView.getContentArea().getSelectedFrame();
4     // if an internal frame is selected
5     if (selectedInternalFrame != null) {
6        // if the selected frame is an EventLogEditor
7        if (selectedInternalFrame instanceof EventLogEditor) {
8           // get the editor
9           EventLogEditor eventLogEditor = (EventLogEditor)selectedInternalFrame;
10          // cut the selected text, if present, into the system clipboard
11          eventLogEditor.cut();
12       }// do nothing if other frame types are selected
13    }// do nothing if there are no selected frames
14 }
```

É importante notare come l'utilizzo delle funzioni taglia, incolla, annulla

e ripeti, nel caso queste effettivamente apportino delle modifiche al contenuto dell'editor, cambia lo stato del contenuto dell'editor, che passa a modificato e non validato: questo avviene sempre attraverso il listener `EventLogEditorDocumentListener`, che rileva ogni modifica al contenuto del documento, quindi anche quelli effettuati da queste funzioni. Questo listener inoltre cancella i dati salvati dalla classe `EventLogEditor` riguardanti i report di analisi precedentemente compiuti, i filtri impostati e i pannelli aperti ad esso associati, in quanto in seguito alla modifica devono essere rigenerati utilizzando il nuovo contenuto modificato. Vedremo meglio questi valori salvati nella sezione dedicata all'analisi dei log di eventi.

4.2.7 Gestione della dimensione del testo: Zoom

Il menù dei pulsanti fornisce un'altra utile funzionalità per la gestione dell'editing dei documenti, ovvero la possibilità di decidere la dimensione del testo utilizzando i pulsanti di *Zoom*.

Attraverso questi pulsanti è infatti possibile incrementare (*Zoom In*), diminuire (*Zoom Out*) e ripristinare al valore di default (*Reset Zoom*) la dimensione del testo all'interno di uno specifico editor.

Nella Figura 4.10 è mostrato un esempio composito di come varia la visualizzazione del documento. Nella metà alta della schermata è visualizzato un editor sul quale è stato fatto uno Zoom Out, mentre in quella bassa è mostrato un editor sul quale è stato fatto Zoom In.

Si può ingrandire o diminuire la dimensione del testo a piacimento, anche molto più piccolo o molto più grande di quello mostrato in Figura 4.10.

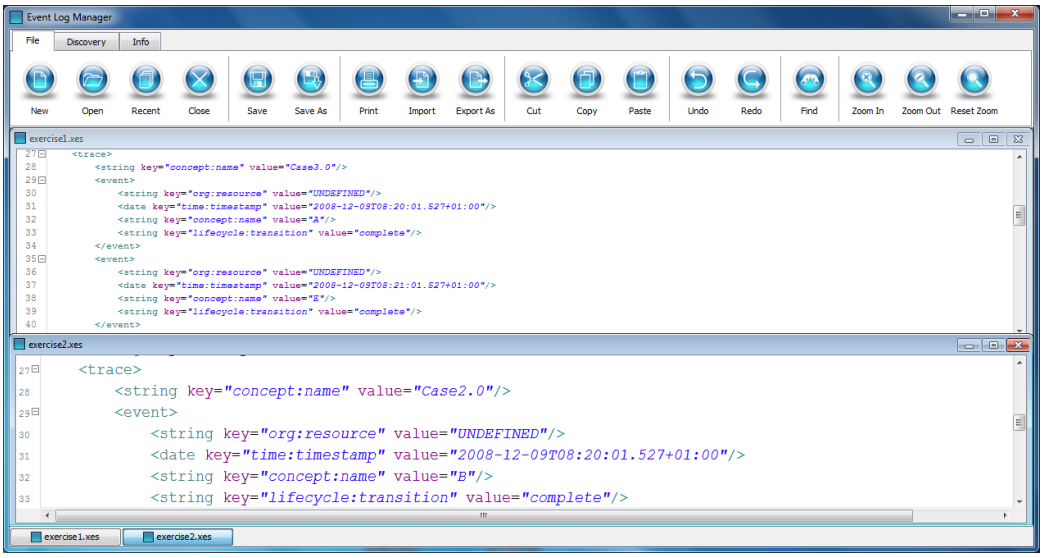

Figura 4.10: La funzione *Zoom*

Capitolo 5

SVILUPPO: GENERAZIONE E ANALISI DEI LOG

In questo secondo capitolo sullo sviluppo dell'applicazione andremo a descrivere le funzioni relative alla generazione e analisi dei log e alla successiva trasformazione sia dei log che dei report di analisi generati in diversi formati.

5.1 Generazione dei log di eventi

In questa prima sezione andremo a vedere una delle funzionalità principali dell'applicazione, ovvero la generazione di log di eventi a partire da diverse soluzioni possibili.

Le diverse possibilità di generazione che andremo a vedere, sono tutte collegate al pulsante "New" dell'interfaccia grafica, di cui in Figura 5.1 ne

viene mostrata l'esecuzione, con la scelta richiesta all'utente riguardo al tipo di generazione che si desidera eseguire.

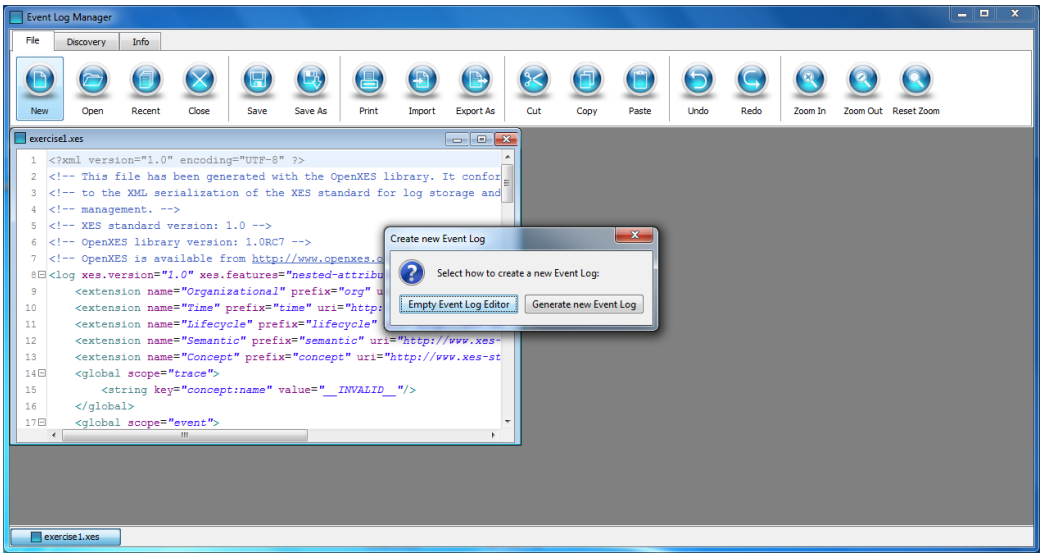

Figura 5.1: La scelta riguardante il metodo di generazione di un nuovo log

Premendo su questo pulsante, il sistema chiederà se si desidera aprire un nuovo editor vuoto dove poter scrivere oppure generare un nuovo log. Entrambi i pulsanti conducono ad una generazione di un log, ma in maniera differente: il primo apre un nuovo editor dove, utilizzando le funzioni che vedremo nella Sezione 5.1.1 e nella Sezione 5.1.2, è possibile generare un nuovo log definendolo passo dopo passo; il secondo invece è connesso alla funzionalità che vedremo nella Sezione 5.1.3, e permette di generare un log in maniera dinamica a partire dalla sua definizione tramite un semplice linguaggio di espressioni.

5.1.1 Le funzioni Template e Completion dell'editor

É venuto il momento di parlare delle altre funzionalità dell'editor dell'applicazione, implementato a partire dalla libreria `RSyntaxTextArea`, descritta nella Sezione 4.2.1.

Questa libreria, oltre alle specifiche già descritte, offre altre due interessanti funzioni che all'interno della nostra applicazione sono state utilizzate per supportare la generazione dei log.

Queste funzioni sono utilizzabili all'interno di un qualsiasi editor del sistema, apribile utilizzando il pulsante "New" dell'interfaccia e scegliendo poi di aprire un nuovo editor (Fig. 5.1).

5.1.1.1 Inserimento automatico del codice: Template

La prima delle due funzioni si chiama *Template*. Questa semplice funzionalità permette di definire una serie di blocchi di codice predefiniti, chiamati appunto template, che possono essere poi richiamati all'interno dell'editor.

Richiamare un blocco di codice significa inserire il suo contenuto all'interno dell'editor in un punto ben preciso per poter così facilmente riscrivere blocchi di codice che vengono utilizzati spesso senza doverli scrivere ogni volta a mano, ma semplicemente richiamandoli.

I template vengono definiti attraverso l'associazione all'editor di una serie di triple, così composte:

1. `id`: l'identificatore univoco del template.
2. `begin`: la prima parte del codice (o tutta) del template.
3. `end`: la seconda parte del codice (oppure anche nessuna) del template.

La suddivisione del blocco tra begin ed end avviene semplicemente in base alla scelta di dove posizionare il cursore dopo l'inserimento del codice all'interno dell'editor: se si vuole il cursore in fondo al blocco del codice, tutto il template va nel campo begin, lasciando il campo end vuoto; se invece si vuole il cursore posizionato in un preciso punto del codice, ad esempio dove si vuole poi inserire un valore, allora si divide il codice in due parti, tra il campo begin e quello end, in modo che le due parti siano inserite rispettivamente prima e dopo il cursore.

Dopo aver definito i template che ci servono, possiamo richiamarli nel seguente modo:

1. Posizionare il cursore nella posizione dove si vuole inserire il template.
2. Scrivere l'identificatore del template.
3. Premere CTRL+SHIFT+SPAZIO

A seguito di questa semplice sequenza di operazioni, il sistema inserirà al posto dell'identificatore del template, il relativo blocco di codice.

Qui di seguito mostriamo come si crea l'associazione tra i template e l'editor, tenendo conto che nel nostro caso i template sono salvati su un file di configurazione esterno ed è il Model ad occuparsi della lettura e scrittura dei dati da questo tipo di file, in quanto è lui il componente dell'MVC adibito a questo compito.

```
1 ...
2 // enable the template function
3 RSyntaxTextArea.setTemplatesEnabled(true);
4 // get the template manager
5 CodeTemplateManager codeTemplateManager = RSyntaxTextArea.getCodeTemplateManager();
```

```
 6 // create the three list for template components
 7 ArrayList<String> id = new ArrayList<String>(templates.get(0));
 8 ArrayList<String> begin = new ArrayList<String>(templates.get(1));
 9 ArrayList<String> end = new ArrayList<String>(templates.get(2));
10 // for every template, identified by an id
11 for (int i=0; i<id.size(); i++) {
12    // add the template to the editor, using id, begin part and end part reading from
          the same index in the three lists
13    codeTemplateManager.addTemplate(new StaticCodeTemplate(id.get(i), begin.get(i), end.
          get(i)));
14 }
15 ...
```

Questa funzione è stata utilizzata nel sistema per fornire all'utente la possibilità di creare o modificare un log all'interno di un editor, senza la necessità di conoscere a fondo la struttura dello standard XES.

Infatti, grazie ad una serie di template preimpostati, uno per ogni possibile elemento strutturale dei log XES, è possibile creare un log completo e complesso senza conoscere nel dettaglio la sintassi degli elementi, ma semplicemente richiamando i vari blocchi di codice e compilandone successivamente i valori, in base alle esigenze.

In Figura 5.2 è mostrato un esempio composito del richiamo di alcuni template all'interno dell'editor.

5.1.1.2 Completamento automatico delle parole: Completion

La seconda funzione dell'editor che andiamo a vedere in questa sezione si chiama `Completion`. Questa funzione permette di definire una lista delle parole chiave che vengono utilizzate all'interno dell'editor in maniera ricorrente

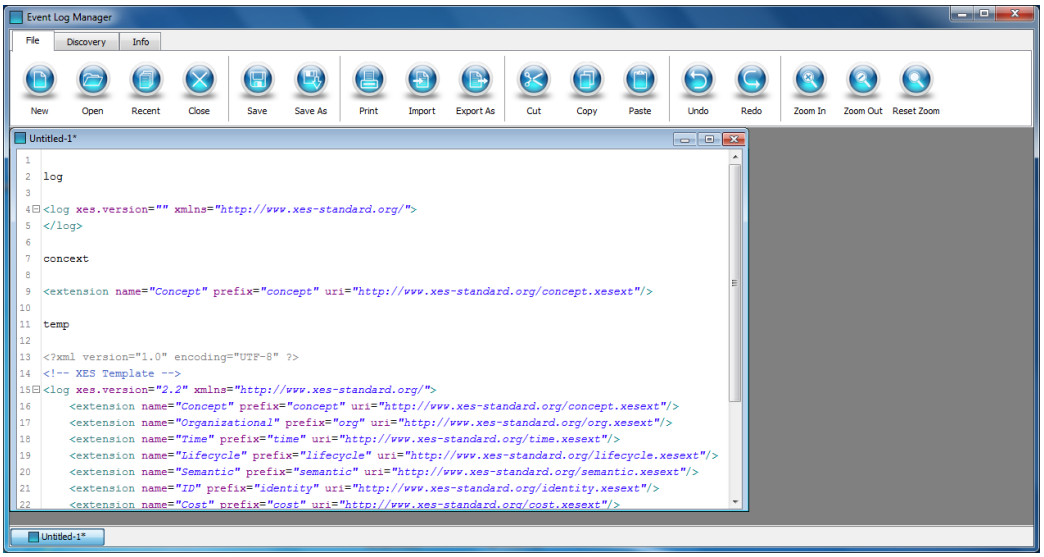

Figura 5.2: L'utilizzo dei `template`

(dipendono da quale è il contenuto dell'editor, nel nostro caso log in formato XES).

Questa lista viene poi associata all'editor che, grazie ad essa, può fornire una funzionalità automatica di completamento delle parole.

```
1 // create the completion provider that will contain the list of possible XES keyword
      completions
2 DefaultCompletionProvider provider = new DefaultCompletionProvider();
3 // add completions reading them from the first String array
4 // a BasicCompletion is a straightforward word completion
5 for (String completion : completions) {
6   provider.addCompletion(new BasicCompletion(provider, completion));
7 }
8 // add the id of all the templates reading them from the second String array
9 for (String templateId : templatesId) {
10   provider.addCompletion(new BasicCompletion(provider, templateId));
11 }
12 // create the auto completion, that will manage the completion in the installed text
      area, using the completion list from the given provider
13 AutoCompletion autoCompletion = new AutoCompletion(provider);
```

```
14 // associate the auto completion with the text area
15 autoCompletion.install(textArea);
```

Anche in questo caso, la lista delle parole chiave è contenuta all'interno di un file di configurazione esterno gestito dal modello. É il Controller che si preoccupa di recuperare tale lista e di associarla ad un editor, quando ne crea uno.

Dopo aver associato la lista delle parole chiave all'editor, vengono inseriti nel completamento automatico anche le chiavi dei template.

In questo modo è possibile utilizzare la funzione di completamento automatico sia per la ricerca delle parole chiave dello standard XES, sia per quella delle chiavi dei template.

Questa funzione può essere richiamata attraverso la combinazione CTRL+SPAZIO:

1. Premendo questa combinazione dopo aver iniziato a scrivere una parola, la funzione apre un menù a tendina mostrando tutte le possibili combinazioni di parole chiave che iniziano con la parte scritta tra cui poter scegliere.

2. Nel caso una sola sia la parola chiave all'interno della lista che inizia con la parte già scritta, la funzione al richiamo tramite tastiera inserirà subito la parola chiave nell'editor, senza aprire alcun menù a tendina, superfluo in questo caso visto che una sola era la parola chiave possibile.

3. Nel caso si richiami la funzione di completamento in un punto vuoto dell'editor, senza alcuna parte iniziale prima della posizione del cursore, questa aprirà il consueto menù a tendina mostrando questa volta la lista di tutte le parole chiave possibili tra cui scegliere, per permettere

una selezione di una intera parola dalla lista senza doverne per forza ricordare la parte iniziale.

In Figura 5.3 viene mostrato il menù a tendina che si apre dopo aver richiamato la funzione di completamento automatico all'interno dell'editor.

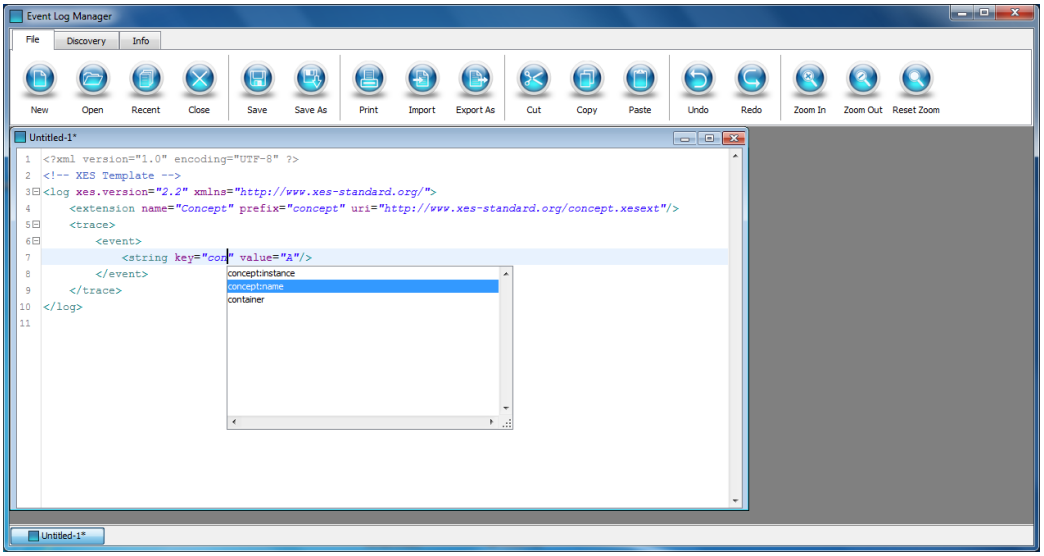

Figura 5.3: La funzione di completamento automatico

Questa funzione si integra con quella della sezione precedente per fornire all'utente uno strumento completo per la generazione dei log.

Mentre l'altra permette di inserire blocchi di codice che definiscono elementi dello standard e che poi devono essere compilati con i valori desiderati, questa funzione è ottima sia per creare o modificare un singolo elemento della struttura, iniziando a scrivere e utilizzando il completamento automatico per non commettere errori nella definizione dei valori da utilizzare, sia soprattutto per la compilazione di tutti i valori all'interno dei template e per la ricerca delle loro chiavi in maniera rapida.

La funzione infatti contiene la lista di tutti gli elementi della versione XML di XES, sia per quanto riguarda i tag (ad esempio "classifier"), sia per quanto riguarda gli attributi dei tag (ad esempio "prefix"), sia per quanto riguarda le chiavi degli attributi definiti dalle estensioni standard da utilizzare all'interno del campo `key` dei tag (ad esempio "concept:name").

In questo modo, combinando la funzione Template alla funzione Completion, la generazione di un log diventa semplice ed automatica, minimizzando il rischio di errori e la necessità di una conoscenza profonda della versione XML di XES, dedicandosi così esclusivamente alla definizione del log da creare.

5.1.2 Definizione di una sintassi per la generazione dei log

Per quanto riguarda la generazione dei log, il sistema fornisce anche altri metodi per raggiungere tale scopo, senza necessariamente la conoscenza del codice XML come abbiamo visto invece nella Sezione 5.1.1.

Infatti l'utilizzo del formato XML, seppur semplificato grazie alle funzioni presentate nella sezione precedente, prevede comunque una conoscenza della struttura XML di un log XES, in modo da costruire il documento nel rispetto dello standard, sia per quanto riguarda la corretta posizione ed utilizzo dei tag XML, sia per quanto riguarda l'utilizzo delle chiavi e dei valori dei vari attributi.

Per permettere all'utente di definire un log in linguaggio più naturale, è stata creata una sintassi per la definizione della struttura di un log, che verrà

poi processata e trasformata dal sistema in un documento XML, senza che quindi l'utente ne conosca nel dettaglio la forma.

La sintassi si basa sul formato CSV per la definizione del log da generare e va inserita all'interno di un editor vuoto dell'applicazione.

Quindi anche in questo caso, come nel precedente, si utilizza il pulsante "New" dell'interfaccia e si sceglie di aprire un nuovo editor (Fig. 5.1). Al suo interno potremo a questo punto utilizzare la sintassi che andremo adesso a vedere.

```
1 trace:id,concept:name,org:resource,time:timestamp
2 1,A,Luca,2015-07-19T17:01:56.399+02:00
3 1,B,Giovanni,2015-07-19T17:01:57.399+02:00
4 1,C,Marco,2015-07-19T17:01:58.399+02:00
5 1,D,Francesco,2015-07-19T17:01:59.399+02:00
6 2,A,Luca,2015-07-19T17:02:00.399+02:00
7 2,B,Giovanni,2015-07-19T17:02:01.399+02:00
8 2,D,Francesco,2015-07-19T17:02:02.399+02:00
```

Seguendo l'esempio appena mostrato e le specifiche del formato CSV, la prima riga non vuota del documento dovrà contenere i nomi delle colonne, separate da virgola.

Il primo di questi nomi sarà `trace:id` e conterrà gli identificatori delle tracce del log. Vedremo in seguito come questa colonna viene utilizzata, per ora ci limitiamo a definirne il nome.

Tutti i successivi nomi devono corrispondere ad una chiave di un attributo di una delle estensioni standard di XES, viste precedentemente. Ad esempio, potremo avere come nomi colonna `concept:name` e`org:resource`, che indicano rispettivamente l'attributo che memorizza il nome di un evento e quello che ne memorizza la risorsa che lo ha eseguito, oppure il nome

colonna `time:timestamp`, che registra il momento di esecuzione dell'evento.

Nelle successive righe del documento vengono inseriti i valori per queste colonne, come da formato CSV.

Ogni riga corrisponde ad un evento del log che stiamo generando, dove il primo valore della colonna `trace:id` corrisponde alla traccia alla quale questo evento appartiene, mentre i successivi valori corrispondono ai valori degli attributi definiti nella riga delle colonne, tutti ovviamente separati da virgola.

Ogni evento dunque sarà rappresentato da una riga nel documento, indicherà la traccia di appartenenza e i suoi valori per gli attributi elencati nei campi colonna, se presenti. Nel caso infatti un evento non abbia un dato valore per l'attributo specificato, questo può essere lasciato vuoto.

Ogni traccia di conseguenza sarà rappresentata da un identificatore e da un insieme di eventi, ovvero le righe del documento che riportano il suo identificatore come primo elemento.

Procedendo con questa sintassi si raggiunge la definizione di un log, che sarà costituito quindi da tutte le tracce ed eventi elencati nel documento, con questi ultimi contenenti a loro volta gli attributi per i quali hanno specificato un valore.

Una volta definito un log in questo modo, si può procedere alla generazione del log vero e proprio in formato XML.

Questa operazione può essere eseguita attraverso il pulsante "Generate" presente nel tab "Discovery" dell'interfaccia (Fig. 4.2).

Questo comando utilizza per la trasformazione del documento CSV in un log XES una classe `CsvXesParser` di cui abbiamo già parlato nella Sezio-

ne 3.3.4.3.

Questa classe come già accennato estende la classe `XParser` delle librerie OpenXES per definire un parser XES a partire da un documento scritto con la sintassi spiegata precedentemente.

Per prima cosa il parser ricerca la prima riga non vuota del documento, che come sappiamo contiene i nomi delle colonne.

Se riesce a trovarla (ovvero il documento non è vuoto), legge i nomi delle colonne e li valida, controllando che questi appartengano all'insieme degli attributi delle estensioni standard di XES.

A questo punto il parser inizia a leggere ogni riga del documento, generando passo dopo passo un oggetto `XLog` contenente tante tracce `XTrace` quanti sono gli identificatori che le rappresentano e contenenti tanti eventi `XEvent` quante sono le righe associate allo stesso identificatore.

```
1  ...
2  // create a XLog
3  XLog log = xesFactory.createLog();
4  ...
5  String [] values;
6  // we start reading all the other lines to convert them in XES elements
7  while ((values = reader.readNext()) != null) {
8     // update the counter
9     lineNumber++;
10    // if the line is empty, skip to next line
11    if (values[0].trim().isEmpty()) {
12       continue;
13    }
14    ...
15    // if we are looking at the first trace in the file or to a new one
16    if (traceID == null || !values[0].equals(traceID)) {
17       // update the trace id
18       traceID = values[0];
```

```
19
20      // create a valid XID based on the passed id
21      XID traceXID;
22      try {
23         // transform the id into a XID
24         traceXID = XID.parse(traceID);
25      } catch(RuntimeException e) {
26         // otherwise generate a new random XID
27         traceXID = new XID(UUID.randomUUID());
28      }
29      // if this XID has already been generated for a trace in this log
30      while (traceXIDList.contains(traceXID)) {
31         // get another XID and try to check again
32         traceXID = new XID(UUID.randomUUID());
33      }
34      // add the unique XID to the xidList
35      traceXIDList.add(traceXID);
36      // clear eventXID list because we have a new trace now
37      eventXIDList.clear();
38      // create the id attribute for the trace
39      XAttributeID traceXIDAttribute = xesFactory.createAttributeID("identity:id",
             traceXID, null);
40      // create the trace attribute map
41      XAttributeMap traceMap = xesFactory.createAttributeMap();
42      // add the id attribute to the map
43      traceMap.put("identity:id", traceXIDAttribute);
44      // create a new trace, using the attribute map just defined
45      trace = xesFactory.createTrace(traceMap);
46      // add the trace to the log
47      log.add(trace);
48   }
49   // create and event attribute map
50   XAttributeMap eventMap = xesFactory.createAttributeMap();
51   // add to the map the attributes that correspond to the column names
52   eventMap.putAll(attributeMap);
53   // fix a limit based on the number of the values in this line
54   int limit = values.length;
```

```
55  // for every column name in the column line
56  for (int i=1; i<columns.length; i++) {
57     // get the attribute that corresponds to this column
58     XAttribute attribute = eventMap.get(columns[i]);
59     // get the value to add to this attribute
60     String value;
61     // if the column index is lower than the value index (for this column there is a
           value) and if the value is non empty
62     if ((i < limit) && (!values[i].trim().isEmpty())) {
63        // read the value corresponding to this column
64        value = values[i];
65     } else {
66        // remove the attribute from the event map because for this event this
              attribute is not defined
67        eventMap.remove(columns[i]);
68        // go to next column
69        continue;
70     }
71     boolean error = false;
72     // depending on the attribute type, add the value to this attribute (and if an
           error occurs, show it)
73     if (attribute instanceof XAttributeLiteral) {
74        try {
75           ((XAttributeLiteral)attribute).setValue(value);
76        } catch(RuntimeException e) {
77           error = true;
78        }
79     } else if ...
80     // if an error occurred while setting the value
81     if (error) {
82        ....
83     }
84  }
85  // if there are more values than column names, we report this to the user
86  if (columns.length < values.length) {
87     // set a warning
88     String warning = "Line "+lineNumber+" has "+(values.length-columns.length)+" more
```

```
                value/s. \n\n";
89        warnings.append(warning);
90      }
91      // create an event using the attribute map just defined
92      XEvent event = xesFactory.createEvent(eventMap);
93      // add the event to its trace
94      trace.add(event);
95  }
96  ...
```

Per ognuno di questi eventi, il parser aggiunge a questi solo gli attributi per i quali vi è un valore, ignorando gli altri (una riga del tipo "1,A„Francesco" indica che per il terzo attributo non vi è valore per questo evento).

Questi valori vengono prima controllati per verificare che siano conformi al tipo che l'attributo definito dalla colonna richiede: passare il valore "Francesco" nella colonna `time:timestamp` non è corretto e il parser fermerà la sua esecuzione notificando all'utente l'errore riscontrato.

Ad ogni traccia generata inoltre, il parser imposta un identificatore conforme allo standard XES a partire da quello passato nel documento: se il valore passato è già conforme allo standard, viene utilizzato quello, altrimenti ne viene generato uno a partire da quello passato che viene trasformato in uno `XID` valido.

Il parser tiene conto di eventuali righe vuote all'interno del documento, saltandole.

Inoltre, gestisce sia il caso di righe contenenti meno valori rispetto al numero di colonne, sia il caso opposto. Per il primo caso, il parser assume come da standard CSV che l'evento definito dalla riga in esame non abbia un valore definito per le colonne in eccesso e quindi non aggiunge tali attributi a

quell'evento, non avendo per questi un valore da utilizzare; nel secondo caso, quando il numero di colonne è minore dei valori indicati nella riga, il parser dapprima ignora i valori in eccesso, in quanto non vi sono colonne che ne definiscono l'attributo a cui appartengono, successivamente, alla generazione del log XES, informerà l'utente con un avviso che alcune righe contenevano più valori di quelli definiti e che questi valori sono stati scartati.

Il procedimento di generazione termina con l'apertura di un `EventLogEditor` contenente il log generato in formato XML.

In Figura 5.4 è mostrato a sinistra un editor contenente l'esempio utilizzato in questa sezione, a destra il log da esso generato.

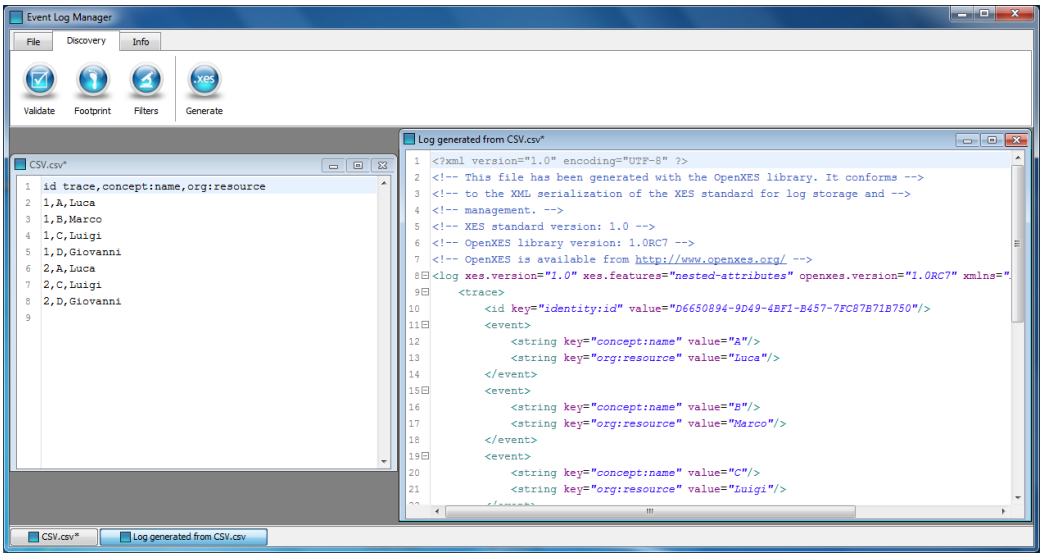

Figura 5.4: La generazione di un log a partire da una sintassi CSV

5.1.3 Generazione di log a partire da espressioni

Per generare un nuovo log il sistema fornisce non soltanto le funzioni Template e Completion e la sintassi in formato CSV da utilizzare all'interno di un editor, ma anche una terza funzionalità: la generazione tramite espressioni algebriche.

5.1.3.1 Come si definisce l'espressione

Definiamo innanzitutto cosa sono queste espressioni e come sono composte.

Una espressione per la generazione di un log è una formula che definisce, attraverso specifici elementi, un insieme di possibili sequenze di eventi (in altre parole un insieme di tracce) che si possono generare a partire da quest'ultima.

Di seguito un esempio di una semplice espressione:

```
1 ( e1 e2 e3 | e4 e5 ) + ( e6 | (e7 e8 + e9) )
```

Questa espressione può essere composta dai seguenti elementi:

1. Una stringa alfanumerica (composta quindi dai soli caratteri A-Z, a-z e 0-9) di lunghezza a piacere senza spazi al suo interno definisce un evento (ad esempio l'elemento "e1").

2. Una sequenza di stringhe alfanumeriche separate tra di loro da uno spazio definisce un insieme di eventi tra loro collegati ed in uno specifico ordine di esecuzione (ad esempio la sequenza "e4 e5").

3. Gli operatori utilizzabili sono due, l'operatore somma o unione (definito dal carattere "+") e l'operatore di `interleaving` (definito dal carattere

"|") che possono essere utilizzati sia tra due singoli eventi che tra due insiemi di eventi. Nelle espressioni l'operatore di interleaving ha sempre la precedenza su quello di unione

4. Si possono utilizzare un numero a piacere, purché tra loro uguale, di parentesi tonde aperte e chiuse per definire l'ordine delle operazioni tra eventi ed insiemi di eventi, in maniera del tutto simile a come si usano per le espressioni algebriche.

L'espressione come detto definisce le possibili tracce del log: infatti se andiamo ad elaborare l'espressione nell'esempio otteniamo l'insieme delle seguenti tracce:

```
 1 e1 e2 e3 e4 e5
 2 e1 e2 e4 e3 e5
 3 e1 e2 e4 e5 e3
 4 e1 e4 e2 e3 e5
 5 e1 e4 e2 e5 e3
 6 e1 e4 e5 e2 e3
 7 e4 e1 e2 e3 e5
 8 e4 e1 e2 e5 e3
 9 e4 e1 e5 e2 e3
10 e4 e5 e1 e2 e3
11 e6 e7 e8
12 e7 e6 e8
13 e7 e8 e6
14 e6 e9
15 e9 e6
```

Queste tracce altro non sono che tutte le possibili combinazioni degli eventi presenti nell'espressione, calcolate valutando gli operatori tra gli insiemi di eventi e la loro precedenza in base alle parentesi.

5.1.3.2 Generazione delle tracce del log

Dopo aver descritto come scrivere una espressione per la definizione di un insieme di possibili tracce, passiamo adesso a vedere come il sistema passa da questa espressione a generare un log.

Per cominciare, indichiamo che a questa funzionalità si accede utilizzando il pulsante "New" dell'interfaccia e scegliendo questa volta di voler generare un nuovo log (Fig. 5.1).

Questa scelta porta all'apertura della finestra in Figura 5.5, dove viene mostrato il pannello attraverso il quale l'utente può inserire l'espressione e confermare la generazione del log.

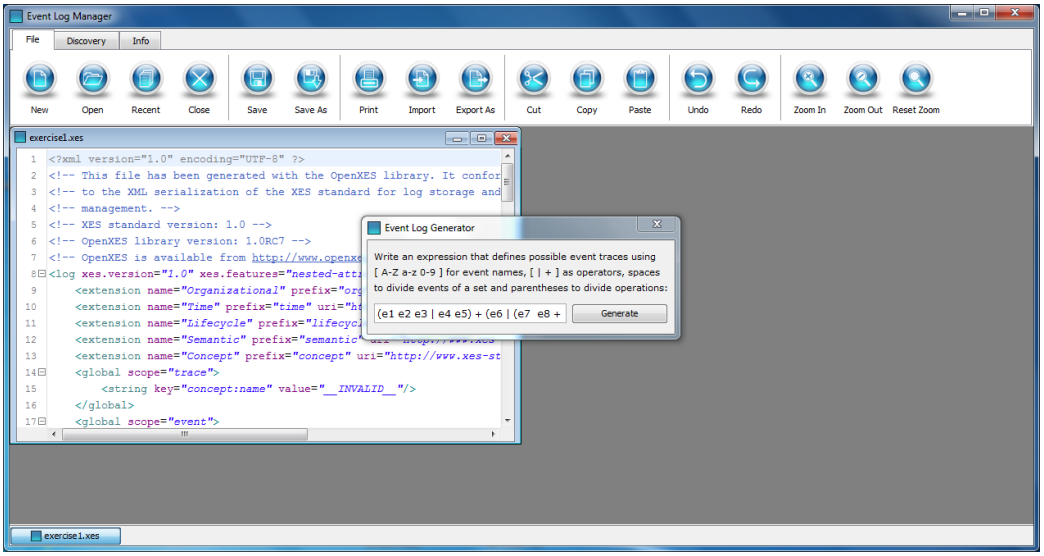

Figura 5.5: La finestra per la generazione dei log tramite espressioni

Dopo aver inserito l'espressione e confermato la generazione, è la classe `EventLogGenerator` del Controller che si occupa della sua elaborazione.

Questo procedimento è diviso in tre semplici passaggi.

La validazione dell'espressione

Per prima cosa la classe controlla che vi sia una espressione da utilizzare (ovvero che l'utente non abbia inserito solo spazi o addirittura nessun carattere), poi verifica il corretto bilanciamento delle parentesi (ovvero che vi siano tante parentesi aperte quanto chiuse), infine si accerta che questa sia composta solo dai caratteri ammessi (come accennato prima, sono validi solamente i caratteri alfanumerici nell'insieme "A-Z a-z 0-9", gli spazi, le parentesi tonde e i simboli "+" e "|" degli operatori).

```
1  // get the expression removing leading and trailing spaces and making multiple spaces
       singles
2  String expression = exp.trim().replaceAll(" +", " ");
3  // if the expression is empty
4  if (expression.isEmpty()) {
5     // throw a specific exception
6     throw new Exception("Invalid expression: no character entered.");
7  }
8  // get the number of left and right parentheses
9  int leftParentheses = expression.length() - expression.replace("(", "").length();
10 int rightParentheses = expression.length() - expression.replace(")", "").length();
11 // if the number of left and right parentheses is not equal
12 if (leftParentheses != rightParentheses) {
13    // get the difference
14    int difference = leftParentheses - rightParentheses;
15    StringBuilder error = new StringBuilder("there");
16    // if the difference is positive, the left parentheses are more than the right ones
17    if (difference > 0) {
18       ...
19    } else { // if negative, the right parentheses are more than the left ones
20       ...
21    }
22    // throw a specific exception
23    throw new Exception("Invalid expression: " + error.toString());
24 }
```

```
25 // if the expression contains character not included in [ A-Z a-z 0-9 + | ( ) ]
26 if (!expression.matches("^[a-zA-Z0-9|+() ]*\$")) {
27    // throw a specific exception
28    throw new Exception(...)
29 }
```

Al termine di questa prima parte, la classe si preoccupa di dividere l'espressione in singoli elementi (mantenendo gli insiemi di eventi uniti come un unico elemento), eliminando eventuali spazi superflui inseriti dall'utente, per poi passare l'espressone così processata al passaggio successivo.

La notazione postfissa

La seconda parte del processo prevede la trasformazione dell'espressione che l'utente inserisce nella naturale notazione infissa (quella utilizzata comunemente per la scrittura delle espressioni algebriche) alla relativa notazione postfissa, chiamata anche *Reverse Polish Notation* (RPN).

Questa notazione permette di eliminare le parentesi dall'espressione, ridistribuendo i rimanenti elementi (ovvero operandi e operatori) in un ordine ben preciso (prima gli operandi e poi gli operatori) per identificare chiaramente l'ordine di esecuzione delle operazioni, anche in assenza di parentesi e mantenendo la precedenza tra i differenti operatori.

Così una espressione infissa del tipo:

```
1 ( a + b ) | ( c + d )
```

Viene trasformata in:

```
1 a b + c d + |
```

La notazione postfissa è molto utilizzata dai calcolatori per i quali risulta difficile elaborare le espressioni infisse, data la loro forma che rimanda le operazioni fino all'ultimo (per via delle parentesi). La notazione postfissa dispone invece operandi ed operatori in modo che il calcolo risulti semplice e diretto per un calcolatore (come vedremo meglio nella prossima sezione).

L'espressione inserita dall'utente viene così in questo passaggio trasformata, in modo da renderla più facilmente calcolabile all'interno dell'applicazione.

Questo viene fatto utilizzando il famoso *Shunting-yard algorithm* di Edsger Dijkstra, che trasforma proprio espressioni dalla notazione infissa a quella postfissa.

```
1  // the output array where to add the elements in postfix notation
2  ArrayList<String> output = new ArrayList<String>();
3  // the stack to use while calculating the postfix notation
4  Stack<String> stack = new Stack<String>();
5  // for every token of the expression
6  for (String token : infixExpression) {
7     // if the token is an operator
8     if (isOperator(token)) {
9        // while stack not empty AND stack top element is an operator
10       while (!stack.empty() && isOperator(stack.peek())) {
11          // if the operator has minor (or equal, in case of left associative)
                 precedence than the operator on top of the stack
12          if ((isAssociative(token, leftAssociative) && getPrecedence(token, stack.peek
                 ()) <= 0) ||
13             (isAssociative(token, rightAssociative) && getPrecedence(token, stack.peek()
                    ) < 0)) {
14             // add to the output the top element of the stack (an operator)
15             output.add(stack.pop());
16             // go to check next top element
17             continue;
18          }
```

```
19          // stop if the top element is not an operator or the stack is empty
20          break;
21       }
22       // push the new operator on the stack
23       stack.push(token);
24    } else if (token.equals("(")) { // else if the token is a left parenthesis
25       // push the parenthesis on the stack
26       stack.push(token);
27    } else if (token.equals(")")) { // else if the token is a right parenthesis
28       // while stack not empty AND stack top element is not a left parenthesis
29       while (!stack.empty() && !stack.peek().equals("(")) {
30          // add the elements from the stack to the postfix output
31          output.add(stack.pop());
32       }
33       // remove the left parenthesis from the stack
34       stack.pop();
35    } else { // if the token is a symbol
36       output.add(token);
37    }
38 }
39 //at the end we move the last remaining elements from the stack to the postfix output
40 while (!stack.empty()) {
41    // add to the postfix output the remaining stack elements
42    output.add(stack.pop());
43 }
44 ...
```

Una volta eseguita la trasformazione, l'espressione viene passata al terzo ed ultimo passaggio del processo di calcolo.

Il calcolo dell'espressione

L'ultimo passaggio del processo prevede a questo punto il calcolo a partire dall'espressione in notazione postfissa dell'insieme di tutte le possibili tracce che l'espressione definisce.

```
1 // a stack where to place elements while calculating the expression
2 Stack<String> stack = new Stack<String>();
3 // for each token of the expression
4 for (String token : postfixExpression) {
5    // if the token is a value
6    if (!isOperator(token)) {
7       // push it onto the stack
8       stack.push(token);
9    } else { // if the token is an operator
10      // pop top two entries
11      String string2 = stack.pop();
12      String string1 = stack.pop();
13      // calculate the result
14      String result;
15      if (token.equals("+")) {
16         result = plus(string1, string2);
17      } else if (token.equals("|")) {
18         result = interleaving(string1, string2);
19      } else {
20         // throw a specific exception
21         throw new Exception("Invalid expression: the operator \""+token+"\" is not
                permitted.");
22      }
23      // push the result onto the stack
24      stack.push(result);
25   }
26 }
27  // pop from the stack the result of the expression (the only value remained inside the
        stack is the result)
28 String result = stack.pop();
29 // return an array composed by all the generated strings (here we calculate the + sign
       as a definition of a set of strings)
30 return result.split("[+]");
```

Un calcolatore può eseguire il calcolo di una espressione in notazione postfissa in maniera molto più semplice rispetto a quello di una infissa, in

quanto questa notazione permette un processo di calcolo lineare, che è così composto:

1. L'espressione viene analizzata da sinistra a destra e si inizia ad inserire gli operandi trovati all'interno di una pila.

2. Ogni volta che si trova un operatore, si toglie dalla pila gli ultimi due operandi inseriti, si effettua il calcolo dell'operatore su questi ultimi e si inserisce il risultato ottenuto di nuovo sulla pila.

3. Si continua quindi ad analizzare l'espressione procedendo come appena descritto, fino a quando all'interno della pila non rimane solo un valore, ovvero il risultato finale.

Come si può vedere, il calcolo è davvero semplice in questa forma e spiega da sé il motivo per il quale abbiamo utilizzato questo procedimento.

Di seguito riportiamo il codice che implementa la funzione di interleaving tra due insiemi di elementi.

```
1  private String interleaving(String string1, String string2) {
2     // get here the result of the interleaving
3     StringBuffer result = new StringBuffer();
4     // split the left and right strings of the interleaving to get all the elements
5     String[] left = string1.split("[+]");
6     String[] right = string2.split("[+]");
7     // interleave all the possible combination of elements of the two strings
8     for (int i = 0; i < left.length; i++) {
9        for (int j = 0; j < right.length; j++) {
10          // add to the final result the generated set of strings using the interleaving
                 operator
11          if (result.length()!=0) result.append("+");
12          result.append(interleaveStrings(left[i], right[j]));
13       }
```

```
14    }
15    // return the final string composed by the sum ("+" between strings) of all the
          generated strings
16    return result.toString();
17 }
18
19 private StringBuffer interleaveStrings(String string1, String string2) {
20    // get the output
21    StringBuffer output = new StringBuffer();
22    // divide the two passed strings in arrays of elements
23    ArrayList<String> array1 = new ArrayList<String>(Arrays.asList(string1.split("\\s+")
          ));
24    ArrayList<String> array2 = new ArrayList<String>(Arrays.asList(string2.split("\\s+")
          ));
25    // interleave the two sets of strings
26    ArrayList<ArrayList<String>> result = interleaveSets(array1,array2);
27    // at the end, for every list in the main list
28    for (int i=0; i<result.size(); i++) {
29       // add a + to connect lists
30       if (i!=0) output.append("+");
31       // get the list
32       ArrayList<String> trace = result.get(i);
33       // transform the list in a string of elements, divided by space
34       for (int j=0; j<trace.size(); j++) {
35          output.append(trace.get(j));
36          if (j != trace.size()-1) output.append(" ");
37       }
38    }
39    // return the generated set of strings
40    return output;
41 }
42
43 private ArrayList<ArrayList<String>> interleaveSets(ArrayList<String> array1, ArrayList
       <String> array2) {
44    // get the output of this interleaving operation
45    ArrayList<ArrayList<String>> output = new ArrayList<ArrayList<String>>();
46    // if the lists are not null, otherwise return an empty list
```

```
47    if (array1 != null && array2 != null) {
48       // if the first list is empty
49       if (array1.size()==0) {
50          // add to the output the second list if not empty
51          if (array2.size()!=0) output.add(new ArrayList<String>(array2));
52       } else if (array2.size()==0) { // else if the second list is empty
53          // add to the output the first list
54          output.add(new ArrayList<String>(array1));
55       } else { // if both lists are not empty
56          // get the first element of list one
57          String prefix = array1.get(0);
58          // the remaining part is the suffix
59          ArrayList<String> suffix = new ArrayList<String>(array1.subList(1, array1.size
                 ()));
60          // calculate the first half of the final output, calling the interleaving
                 method recursively on this subset
61          ArrayList<ArrayList<String>> output1 = interleaveSets(suffix, array2);
62          // append the prefix on top of the generated lists
63          appendAll(prefix,output1);
64          // add the modified partial output to the final one
65          output.addAll(output1);
66          // get the first element of list two
67          prefix = array2.get(0);
68          // the remaining part is the suffix
69          suffix = new ArrayList<String>(array2.subList(1, array2.size()));
70          // calculate the second half of the final output, calling the interleaving
                 method recursively on this subset
71          ArrayList<ArrayList<String>> output2 = interleaveSets(array1,suffix);
72          // append the prefix on top of the generated lists
73          appendAll(prefix,output2);
74          // add the modified partial output to the final one
75          output.addAll(output2);
76       }
77    }
78    // return the generated output
79    return output;
80 }
```

```
81
82 private void appendAll(String s, ArrayList<ArrayList<String>> output) {
83    // for every list
84    for (ArrayList<String> el : output)
85       // add the passed element on top of the list
86       el.add(0,s);
87 }
```

In particolare sottolineando l'utilizzo della ricorsione per il calcolo incrementale delle possibili combinazioni di stringhe.

Al termine del calcolo, la classe `EventLogGenerator` restituisce l'insieme delle tracce generate.

5.1.3.3 Visualizzazione delle tracce all'utente

Dopo aver generato l'insieme delle tracce del log, definite dall'espressione inserita dall'utente, il sistema mostra a quest'ultimo il risultato di tale calcolo.

In Figura 5.6 viene mostrato il messaggio di notifica del sistema a fine generazione.

Attraverso questa finestra l'utente può per prima cosa controllare se le tracce generate rispettano quello che aveva pianificato.

Successivamente può scegliere se scartarle, per modificare l'espressione e generarne di altre, oppure decidere come vuole che queste siano utilizzate per l'ultimo passaggio di questa procedura.

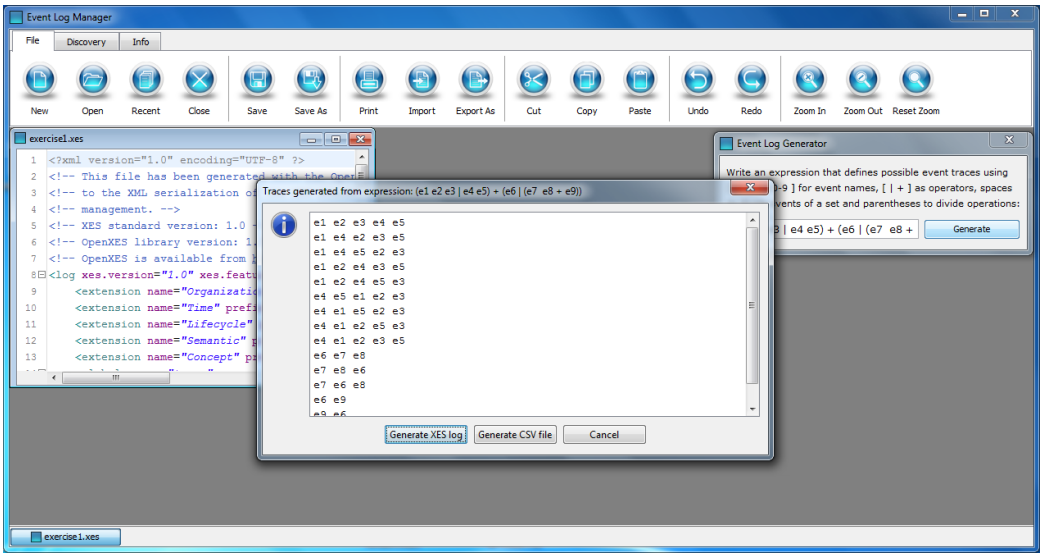

Figura 5.6: Il risultato della generazione delle tracce

La generazione del log XES

Nel caso in cui l'utente selezioni di generare un log XES, quello che otterrà sarà l'apertura di un editor al cui interno sarà inserito in formato XML il log prodotto utilizzando l'insieme delle tracce generate.

Questo procedimento è molto semplice, in quanto abbiamo un insieme di elementi, uno per ogni traccia del log, e per ogni elemento un insieme di nomi di eventi.

All'applicazione è sufficiente creare un oggetto `XLog`, inserire al suo interno una traccia `XTrace` per ogni elemento dell'insieme e inserire all'interno di queste i rispettivi eventi `XEvent`.

A ciascun evento inoltre vengono dati due attributi: il primo, `concept:name`, conterrà il nome dell'evento, quello definito all'interno dell'espressione; il secondo, `time:timestamp`, conterrà il timestamp dell'esecuzione dell'evento.

Questo secondo attributo, essendo un log generato e non un log reale, avrà un valore fittizio: a partire dal timestamp di generazione del log, ogni evento avrà un valore per questo attributo incrementato rispetto all'evento precedente nel log.

Il valore di questo incremento è contenuto all'interno di un file di configurazione, gestito dal modello e modificabile dal pannello delle impostazioni dell'applicazione, di cui parleremo nella Sezione 6.4. Il modello fornisce al Controller il valore attualmente salvato nelle configurazioni dell'applicazione e quest'ultimo lo utilizza come incremento per il timestamp degli eventi.

```
1  // create the log
2  XLog log = xesFactory.createLog();
3  ...
4  // get the current timestamp
5  long timestamp = System.currentTimeMillis();
6  // set the time difference between events
7  int seconds = elmModel.getEventTimestampDistance();
8  // for each generated trace
9  for (String eventTrace : eventTraces) {
10   // create an attribute map for the trace
11   XAttributeMap traceMap = xesFactory.createAttributeMap();
12   ...
13   // add the id to the attribute map
14   XAttributeID traceXIDAttribute = xesFactory.createAttributeID("identity:id", xid,
         null);
15   traceMap.put("identity:id", traceXIDAttribute);
16   // create a trace with the attribute map just set
17   XTrace trace = xesFactory.createTrace(traceMap);
18   // add the trace to the log
19   log.add(trace);
20   // get the event names from the generated trace
21   String[] events = eventTrace.split("\\s+");
22   // for every event name in the generated trace
23   for (String eventName : events) {
```

```
24      // create an attribute map
25      XAttributeMap eventMap = xesFactory.createAttributeMap();
26      // add the attribute name to the map
27      XAttributeLiteral eventNameAttribute = xesFactory.createAttributeLiteral("concept
            :name", eventName, XConceptExtension.instance());
28      eventMap.put(eventNameAttribute.getKey(), eventNameAttribute);
29      // add the attribute timestamp to the map, adding seconds to its execution
30      timestamp = (timestamp + (seconds * 1000));
31      XAttributeTimestamp eventTimestampAttribute = xesFactory.createAttributeTimestamp
            ("time:timestamp", timestamp, XTimeExtension.instance());
32      eventMap.put(eventTimestampAttribute.getKey(), eventTimestampAttribute);
33      // create the event with the attribute map just set
34      XEvent event = xesFactory.createEvent(eventMap);
35      // add the event to the trace
36      trace.add(event);
37    }
38 }
```

La generazione del log in formato CSV

Nel caso in cui l'utente selezioni di generare un log in formato CSV, otterrà l'apertura di un editor con all'interno la versione del log definito dalla sua espressione in formato CSV, seguendo lo standard descritto nella Sezione 5.1.2.

La generazione anche in questo caso è molto semplice. L'applicazione inserisce nella prima riga del documento generato l'elenco dei nomi delle colonne:

```
1 trace:id,concept:name,time:timestamp
```

Dopo questa, viene generata una riga per ogni evento di ogni traccia dell'insieme, riportando sulla prima colonna l'identificatore della traccia a cui

questi eventi appartengono e nelle successive due il nome dell'evento e il suo timestamp, calcolati nello stesso modo descritto nella precedente sezione.

```
1  int traceIndex = 0;
2  // add the column line
3  csv.append("trace:id,concept:name,time:timestamp").append("\n");
4  // get the current timestamp
5  long timestamp = System.currentTimeMillis();
6  // set the time difference between events
7  int seconds = elmModel.getEventTimestampDistance();
8  // get the xsDateTimeConverter to convert timestamp to XES timestamp
9  XsDateTimeConversionJava7 xsDateTimeConverter = new XsDateTimeConversionJava7();
10 // for every trace in the generated set of traces
11 for (String eventTrace : eventTraces) {
12    traceIndex++;
13    // get the event names
14    String[] events = eventTrace.split("\\s+");
15    // for each event name
16    for (String event : events) {
17       // add seconds to this event execution
18       timestamp = (timestamp + (seconds * 1000));
19       // calculate the xsDateTime to use as value for the XES timestamp
20       String xsDateTime = xsDateTimeConverter.format(new Date(timestamp));
21       // add to the document the id of the trace and the value of the event attributes
22       csv.append(traceIndex).append(",").append(event).append(",").append(xsDateTime).
              append("\n");
23    }
24 }
```

In Figura 5.7 vengono mostrati tutti gli elementi di questo procedimento di generazione per espressioni: il pannello di inserimento dell'espressione, il log XES e il log in formato CSV generati da quest'ultima.

Il motivo per il quale vi è anche la possibilità di generare un documento CSV è che, mentre il log XES è immediatamente utilizzabile per operazioni di analisi che vedremo in seguito, presenta difficoltà evidenti nel caso si debba

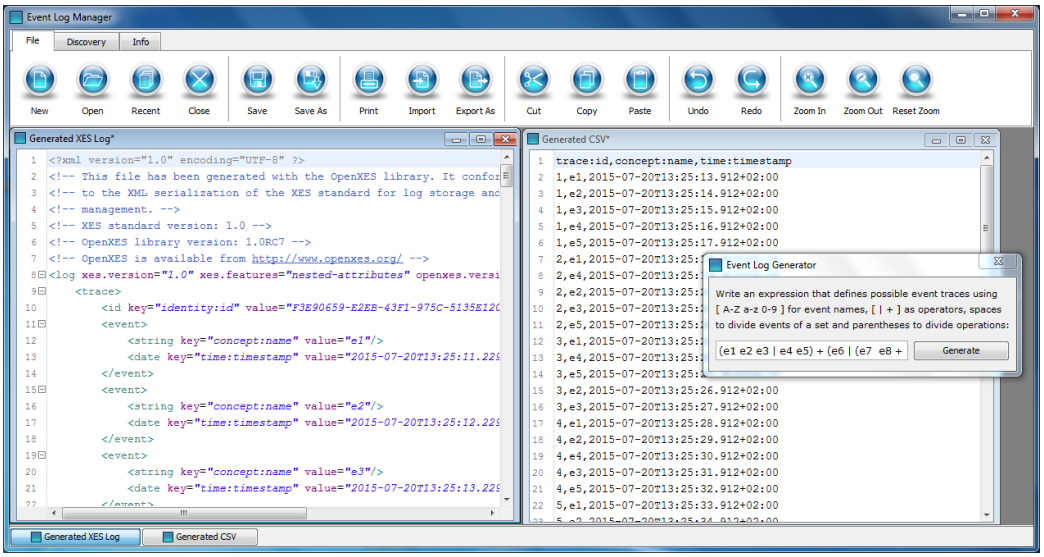

Figura 5.7: Il risultato della generazione attraverso espressioni

ad esempio aggiungere un nuovo attributo a tutti gli eventi del log. Sarebbe necessario scorrere il log riga per riga, ricercando tutti gli eventi ed inserendo per ciascuno di essi una nuova riga contenente il tag relativo a quel dato attributo, con chiave e valore desiderati. Un procedimento certamente non semplice.

Generando invece un documento CSV, sarebbe molto facile inserire un nuovo nome colonna nella prima riga ed altrettanto facile inserire dei valori in quella colonna per ciascun evento del log, in quanto questi sono posizionati uno per ogni riga del documento, ottenendo così un rapido inserimento di un nuovo attributo per ogni evento del log.

A questo punto, dopo aver inserito il nuovo attributo, basterebbe utilizzare la funzione di generazione vista nella Sezione 5.1.2, attivabile attraverso il pulsante "Generate" dell'interfaccia, ottenendo come risultato un log XES con anche il nuovo attributo che in fase di scrittura dell'espressione non era

stato possibile inserire.

5.2 Analisi dei log di eventi

In questa sezione verranno discusse le funzioni dell'applicazione in merito alla validazione ed analisi dei log di eventi.

Prima di parlare delle varie funzioni, verrà completata la descrizione della classe `EventLogEditor`, iniziata nella Sezione 4.2.1. Successivamente verranno descritte le varie funzioni che l'applicazione mette a disposizione per l'analisi dei log.

5.2.1 EventLogEditor e il supporto all'analisi: i filtri

La classe `EventLogEditor`, oltre alle informazioni indicate nella Sezione 4.2.1, può registrarne altre collegate all'analisi del suo contenuto.

Oltre ad informazioni di utilità quali la memorizzazione dell'oggetto `XLog`, rappresentazione Java del log contenuto nell'editor e generato durante l'analisi, e del codice XHTML del report di analisi relativo allo stesso log, entrambi utilizzati per facilitare l'esecuzione delle operazioni di calcolo e visualizzazione, vi sono i filtri.

I filtri sono una serie di impostazioni che l'utente può scegliere e che sono associate ad un dato editor e al suo contenuto.

I filtri permettono di definire determinate prospettive di analisi del log e vengono utilizzati in fase di generazione del report di analisi.

Ogni `EventLogEditor` ha associato un filtro, definito dalla classe `FootprintFilter`. Questa classe definisce una serie di campi relativi a valori da

memorizzare e per ciascuno di essi un insieme di metodi per recuperare (*get*) ed impostare (*set*) questi valori.

I valori che vengono registrati e gestiti dalla classe `FootprintFilter` e che possono essere utilizzati in fase di analisi sono i seguenti:

1. Il numero di occorrenze minime per traccia: questo valore indica quale deve essere il numero minimo di occorrenze di una stessa traccia (costituita quindi da una ben precisa sequenza di eventi) perché questa sia presa in considerazione durante l'analisi del log, o scartata altrimenti.

2. L'insieme dei classificatori: come abbiamo visto nella Sezione 2.1.2.5, lo standard XES permette la definizione di classificatori di eventi, costrutti che raggruppano gli eventi in insiemi in base al valore dei loro attributi; questo valore memorizza l'insieme dei classificatori disponibili per il log analizzato.

3. Il classificatore selezionato: La scelta di un determinato classificatore tra quelli disponibili permette di decidere secondo quali attributi due eventi debbano essere considerati uguali, e quindi cambiare di conseguenza la prospettiva di analisi.

4. Gli eventi iniziali selezionati: si possono definire quali tracce utilizzare per l'analisi e quali scartare, in base ai loro eventi iniziali; in pratica se una traccia inizia con un evento che è stato selezionato, l'analisi prenderà in considerazione quella traccia, altrimenti la scarterà.

5. Gli eventi finali selezionati: come per sopra, si possono definire quali tracce utilizzare per l'analisi e quali scartare, in base ai loro eventi finali;

se una traccia termina con un evento che è stato selezionato, l'analisi prenderà in considerazione quella traccia, altrimenti la scarterà.

Questi possono essere modificati dall'utente utilizzando il pulsante "Filters" dell'interfaccia del menù Discovery.

Nel caso non sia ancora stata effettuata una analisi del log prima dell'apertura del pannello dei filtri, questi ultimi avranno un valore nullo, in quanto non è possibile determinarli a priori senza effettuare una prima analisi del log contenuto nell'editor: considerando che ogni log è diverso, senza analizzarlo è impossibile sapere ad esempio quali siano gli stati iniziali e finali delle tracce o quali siano i classificatori impostati.

In questo caso il sistema chiederà prima se si desidera calcolare i loro valori iniziali per avere un pannello dei filtri completo, oppure se si vuole procedere utilizzando solo il filtro sul numero minimo di occorrenze delle tracce.

Nel primo caso, quello che otterremo sarà una prima analisi del log per il calcolo dei suoi valori iniziali e la successiva apertura di un pannello come quello mostrato in Figura 5.8, implementato utilizzando la classe `FilterPanel`.

Nel secondo caso invece, otterremo un pannello simile ma semplificato, con semplicemente il primo campo (quello sulle occorrenze).

Vedremo meglio nella Sezione 5.2.3 come viene calcolato il valore dei filtri in base al contenuto del log.

I filtri mostrati nel pannello possono ora essere modificati dall'utente, al quale poi si richiede di scegliere tra tre possibili azioni:

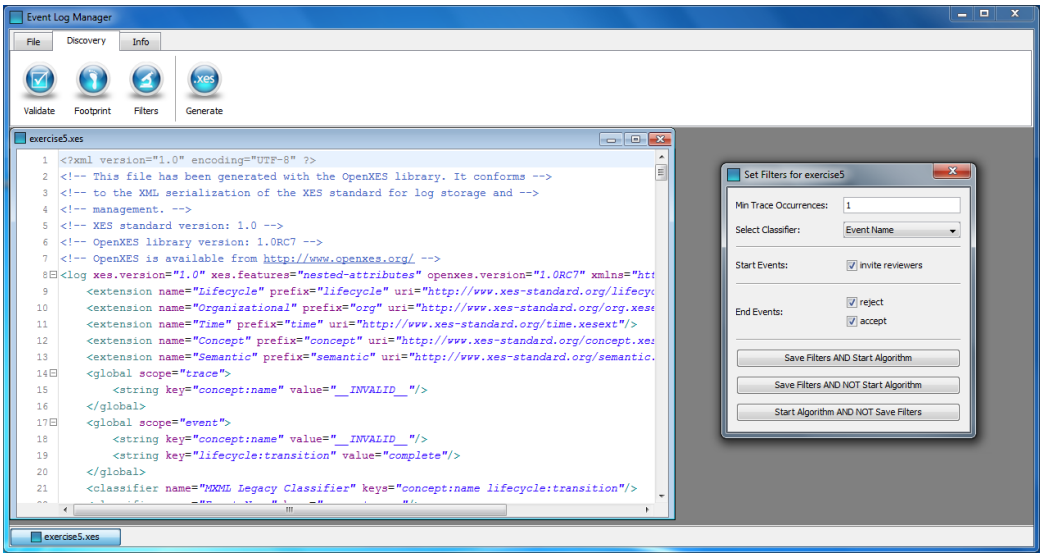

Figura 5.8: Il pannello dei filtri

1. Salvare i filtri impostati nell'oggetto `FootprintFilter` dell'editor e avviare l'algoritmo di analisi del log, che utilizzerà i nuovi filtri impostati.

2. Salvare i filtri impostati nell'oggetto `FootprintFilter` dell'editor ma non avviare l'algoritmo di analisi del log, procedendo solo con l'aggiornamento del filtro associato all'editor selezionato.

3. Avviare l'algoritmo di analisi utilizzando i filtri impostati, ma senza salvarli nell'oggetto `FootprintFilter` dell'editor, in modo da mantenere i filtri salvati ma al contempo effettuare analisi con filtri temporanei diversi da quelli dell'editor.

É importante notare come questo pannello dei filtri sia direttamente collegato con l'editor a cui è associato: se un editor ha un pannello dei filtri aperto, qualsiasi operazione di selezione, minimizzazione, massimizzazione e

chiusura che avvengono sulla finestra dell'editor, hanno un effetto identico anche sul suo pannello dei filtri.

Sottolineiamo infine una importante proprietà della classe `EventLogEditor`, per quanto concerne l'analisi dei log.

Abbiamo notato precedentemente come un editor possa memorizzare l'oggetto `XLog` e il report generato dall'analisi, nonché tenere traccia dei filtri impostati da un utente e definiti sul contenuto del log.

Nel caso avvenga una modifica del contenuto dell'editor, non accade solamente quanto riportato nella Sezione 4.2.1 in merito allo stato di modificato e validato, ma vengono anche cancellati i riferimenti all'oggetto `XLog` e al relativo report generato e azzerati i valori dei filtri calcolati, in quanto sono tutti valori collegati al vecchio contenuto dell'editor e non più validi adesso (dovranno essere rigenerati dal sistema alla prossima analisi).

Dopo aver descritto questa importante funzionalità della classe `EventLogEditor`, possiamo ora passare a vedere come si sviluppa il processo di analisi di un log.

5.2.2 Validazione di un log XES

La prima parte dell'analisi passa attraverso il processo di validazione del log di eventi.

Questa procedura può essere avviata separatamente all'analisi, utilizzando il pulsante "Validate" del menù Discovery, oppure verrà eseguita in automatico se si avvia l'analisi dal pulsante "Footprint" ed il documento risulta non validato.

Il processo di validazione prende il contenuto dell'`EventLogEditor` selezionato, controlla che questo sia salvato (in caso contrario chiede alla funzione di salvataggio di provvedere alla gestione di questa operazione)

Dopodiché procede alla validazione rispetto allo Schema dello standard XES ("xes.xsd") e delle estensioni di XES (xesext.xsd).

```
1  // create a SAXParserFactory to generate the parser
2  SAXParserFactory parserFactory = SAXParserFactory.newInstance();
3  // set namespace aware to receive element names
4  parserFactory.setNamespaceAware(true);
5  // set the Schema for validation
6  parserFactory.setSchema(schema);
7  // create the SAXParser
8  SAXParser parser;
9  try {
10    parser = parserFactory.newSAXParser();
11 } catch (ParserConfigurationException e) {
12    elmView.showErrorMessage("Unable to start the XES Parser. Please try again.");
13    return docValidated; // false
14 } catch (SAXException e) {
15    elmView.showErrorMessage("Unable to start the XES Parser. Please try again. \n"
16                    + "Error: " + e.getMessage());
17    return docValidated; // false
18 }
19
20 // create an instance of the parser content handler
21 ParserContentHandler parserContentHandler = new ParserContentHandler();
22 // parse the file using the handler to manage the SAXException exceptions
23 try {
24    // set the application in a busy state in the meanwhile
25    elmView.setBusyState(true);
26    parser.parse(new File(eventLogEditor.getAbsolutePath()), parserContentHandler);
27    elmView.setBusyState(false);
28 } catch (SAXException e) {
29    /* all the SAXException exceptions are managed by the content handler, this
30     * if needed will throw a SAXException to activate this catch and stop the
```

```
31     * validation*/
32    elmView.setBusyState(false);
33    return docValidated; // false
34 } catch (IOException e) {
35    elmView.setBusyState(false);
36    elmView.showErrorMessage("Unable to read the XES log. Please try again.");
37    return docValidated; // false
38 }
39 // if the parsing process did not throw a fatal error, get the result of the parsing
40 docValidated = parserContentHandler.isDocumentValid();
41 // if the document has been validated, set the content as validated
42 if (docValidated) {
43    eventLogEditor.setContentValidated(true);
44 }
```

Per prima cosa controlla se la connessione Internet è presente. In tal caso recupera tramite il modello i due URL indicati nel file di configurazione (e modificabili dalle impostazioni dell'applicazione, Sez 6.4) che identificano i due Schema online (di default sul sito dello standard XES).

Altrimenti, sempre tramite il modello, recupera il percorso locale dei due Schema e li carica direttamente dal sistema, senza bisogno di accedere ad Internet. Anche questi due valori sono modificabili dalle impostazioni dell'applicazione, come vedremo meglio nella Sezione 6.4.

Una volta caricati correttamente i due Schema, il processo effettua una validazione del log rispetto a questi ultimi.

Nel caso la validazione abbia successo, un messaggio di notifica viene inviato all'utente, riportando il buon esito dell'operazione.

Nel caso invece la validazione abbia riscontrato un errore grave, il processo dichiarerà il log non valido secondo lo standard ed indicherà all'utente dove è stato trovato l'errore nel log e quale esso sia.

In Figura 5.9 è mostrato il caso più interessante in cui il log non sia valido rispetto allo Schema di XES e quindi il sistema segnali l'errore all'utente, indicando i dettagli dell'errore riscontrato.

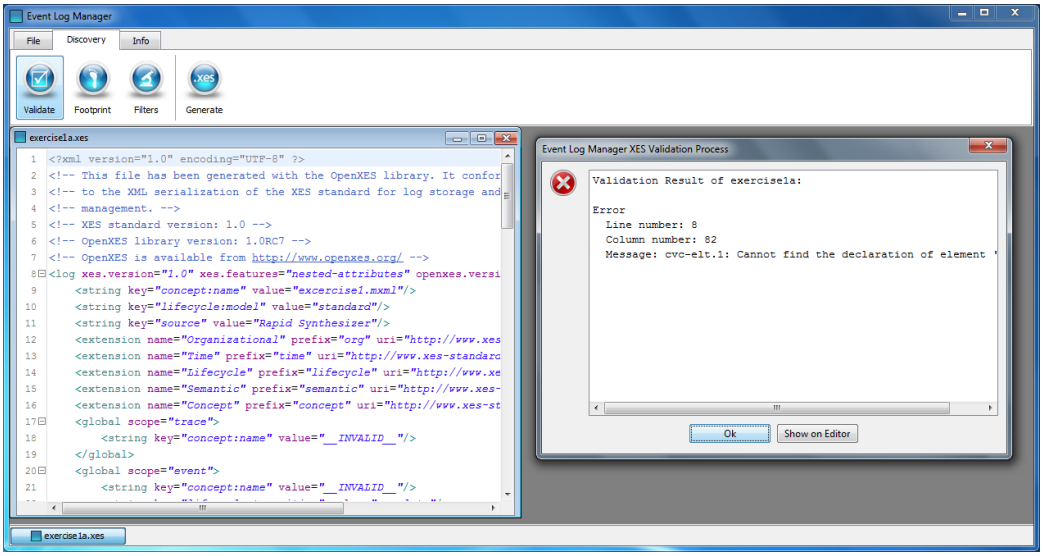

Figura 5.9: Il risultato negativo della validazione

5.2.3 Analisi di un log XES

Dopo una validazione positiva del log, il sistema può finalmente analizzare correttamente il documento.

Questa analisi come già accennato, può essere avviata a partire dal pulsante "Footprint" del menù Discovery dell'applicazione.

Il risultato di questa analisi sarà un report scritto in formato XHTML, visualizzato all'interno di una finestra `FootprintMatrixInfo` e così costituito:

1. La lista delle tracce del log che sono state analizzate, raggruppate per occorrenza.

2. La footprint matrix del log, ovvero la matrice che mostra le dipendenze causali tra gli eventi del log.

3. I filtri utilizzati per la generazione della matrice.

4. La legenda della matrice, dove vengono indicati gli alias utilizzati nelle colonne della matrice e le rispettive classi di eventi che questi alias identificano.

Il procedimento di analisi di un log inizia con la trasformazione dello stesso dal formato XML presente nell'editor selezionato in un oggetto Java `XLog`, utilizzando la classe `XmlXesParser` messa a disposizione dalla libreria OpenXES.

```
1 // create the XES XML Parser
2 XesXmlParser parser = new XesXmlParser(new XFactoryBufferedImpl());
3 // create the destination of the parsing process
4 Collection<XLog> logs = null;
5 // if the file is a .xes file (the XES parser can handle only .xes files)
6 if (parser.canParse(editorFile)) {
7    try {
8       // parse the XML document into a XLog
9       logs = parser.parse(editorFile);
10   } catch (Exception e) {
11      elmView.setBusyState(false);
12      elmView.showErrorMessage("Unable to analyse the document due to XES parsing
            errors. Please try again.");
13      return isGenerated;
14   }
15 }
```

Questa libreria implementa un parser dal formato XML di un log alla sua rappresentazione Java.

Dopo aver ottenuto questo oggetto, l'algoritmo agisce in due modi differenti a seconda se per l'editor selezionato, i filtri siano già stati calcolati (e quindi utilizzabili) oppure siano ancora da definire (e quindi vadano calcolati durante questa analisi).

Nel caso in cui questa sia la prima analisi del log, vengono eseguite le seguenti due attività:

1. Recupero durante l'analisi del log degli elementi che costituiscono i valori iniziali del suo filtro.

2. Analisi del log e generazione di un report di analisi utilizzando il filtro iniziale.

Nel caso in cui invece ci sia già un filtro definito, l'algoritmo deve semplicemente utilizzare proprio quel filtro per effettuare l'analisi del log, invece di quello iniziale.

La prima cosa quindi che il procedimento di analisi fa è capire se ci sia bisogno di completare il filtro del log oppure questo sia già completo ed utilizzabile.

Nella prima situazione, il processo fa una operazione in più, ovvero legge per prima cosa dal log l'insieme dei classificatori definiti e l'insieme degli eventi iniziali e finali di tutte le tracce.

Adesso è in grado di definire il filtro iniziale di questo log, che sarà composto da:

1. Il numero di occorrenze per traccia sarà uguale all'unità, in quanto è un valore indipendente dal log e viene modificato solamente dal pannello dei filtri (quindi in questa analisi non verrà applicata una selezione per occorrenze di traccia).

2. L'insieme dei classificatori definiti all'interno del log, che non servono per l'analisi, ma vengono utilizzati per essere mostrati all'utente nel pannello dei filtri, per mostrare le possibili alternative.

3. Il classificatore selezionato che verrà utilizzato per l'analisi, che equivale sempre al classificatore sull'attributo `concept:name` per quanto riguarda la prima analisi di un log, non avendo ancora ricevuto la scelta da parte dell'utente.

4. L'insieme degli eventi iniziali selezionati, costituito da tutti gli eventi iniziali possibili per il log analizzato (ovvero nessuna selezione per gli eventi iniziali viene eseguita in questa analisi iniziale).

5. L'insieme degli eventi finali selezionati, costituito da tutti gli eventi finali possibili per il log analizzato (ovvero nessuna selezione per gli eventi finali viene eseguita in questa analisi iniziale).

Dopo questa analisi preliminare, l'algoritmo ha in entrambi i casi un filtro completo da utilizzare e procede come segue.

Per ogni traccia del log, il processo controlla se la traccia inizia o finisce con un evento che è indicato come non selezionato per l'analisi. In questo caso la traccia viene scartata e il processo passa alla successiva.

```
1 // get the first event of the trace
```

```
2  XEvent borderEvent = trace.get(0);
3  String attributeValue = "Undefined";
4  // get the name of the event
5  if (borderEvent.hasAttributes()) {
6     XAttributeMap attributeMap = borderEvent.getAttributes();
7     // the name of the event is in the concept:name attribute
8     if (attributeMap.containsKey("concept:name")) {
9        // get the value of this attribute
10       XAttributeLiteral attribute = (XAttributeLiteral)attributeMap.get("concept:name")
              ;
11       attributeValue = attribute.getValue();
12    }
13 }
14 // if the filter is complete
15 if (filterComplete) {
16    // we check if this start event is selected or not
17    Boolean isSelected = startEvents.get(attributeValue);
18    // if it is not selected, skip this trace
19    if (isSelected != null) {
20       if (!isSelected) {
21          continue;
22       }
23    } else {
24       continue;
25    }
26 }
```

Questa è la prima selezione basata sul filtro impostato: la selezione per eventi iniziali e finali delle tracce da analizzare.

Dopo questa fase, avendo inoltre un classificatore da utilizzare definito dal filtro selezionato, il processo ha anche un insieme di attributi definiti dal classificatore che a loro volta permettono di creare classi di eventi basate sui valori di questi attributi.

Grazie a questi attributi può prendere ogni evento di ogni traccia e

classificarlo, in base al valore di questi attributi.

Facendo ciò l'algoritmo ottiene un insieme di classi di eventi, ai quali a ciascuna di esse da un alias alfabetico.

A questo punto per ogni evento di ogni traccia valida, il processo controlla se questo attributo appartiene o meno ad una delle classi generate dal classificatore: in caso affermativo aggiunge l'evento alla lista degli eventi della traccia, altrimenti lo scarta in quanto evento non classificabile.

Questa è la seconda selezione basata sul filtro: la selezione degli eventi per classificazione.

É proprio in questo punto inoltre che il classificatore gioca un ruolo fondamentale per l'analisi del log. Una sequenza di eventi di una traccia ha un ordine ben preciso e definisce di conseguenza un ordine anche delle classi di eventi ai quali questi eventi appartengono. É chiaro dunque che classificare gli eventi per uno o un alto attributo, cambia le classi di eventi generate e quindi cambia l'ordine in cui queste classi sono in relazione tra loro.

Dopo aver ottenuto, a seguito di questi passaggi, una lista di tracce selezionate contenenti a loro volta un insieme di eventi selezionati dal classificatore e definiti dalla loro classe di eventi, il processo può passare ad applicare l'ultimo elemento del filtro selezionato: la selezione delle tracce che appaiono un numero minimo di volte all'interno del log.

L'algoritmo di fatto elimina dalla lista tutte le tracce che hanno un numero di occorrenze minore di quello definito all'interno del filtro.

A questo punto, a filtro completamente applicato, il processo può passare alla fase finale di calcolo delle dipendenze causali tra le classi di eventi generate.

É infatti chiaro che presa una traccia della lista, la sequenza di eventi rimasti sono tutti associati ad una delle classi di eventi generate dal classificatore.

Il processo può quindi sostituire nella lista gli eventi con l'alias della loro classe, rendendo in questo modo evidenti le dipendenze causali tra una e un'altra classe dell'insieme.

L'algoritmo in questa fase crea una matrice N x N, dove N è uguale al numero di classi di eventi per il quale stiamo controllando le dipendenze.

Prendendo di nuovo la lista delle tracce, il processo inizia per ciascuna di esse a scorrere la relativa sequenza di classi di eventi, andando ad aggiungere nella matrice per ogni coppia di elementi (A,B), dove A è l'elemento che precede e B è l'elemento che segue nella sequenza, un valore pari al numero di occorrenze della traccia che li contiene (tale valore dunque è anche il numero di volte in cui tale dipendenza appare nella traccia), proprio nella posizione di indici pari alla coppia (A,B).

```
1  // get the trace events.
2  // event = the set of attributes values read from an event, that defines an event class
3  String[] events = trace.getElement().split(",");
4  // if the set of values has something to use
5  if (events.length > 0) {
6     /* here we not only add the event to the trace list, using the alias
7      * of its respecting class, but we also upgrade the matrix with the
8      * event dependencies for being able to create the footprint matrix */
9     Integer row;
10    // we already fix the first element to use with the id of the class of the first
          trace event
11    Integer col = eventClassIdentifiers.get(events[0].trim());
12    if (col != null) {
13       // add to the trace list the first event, using the alias of its class name
14       traceList.append(eventClassAliases[col]);
```

```
15      // for every other event in the trace
16      for (int i = 1; i < (events.length); i++) {
17         // we move the column value to row, to add now the relation between this event
                and the next one
18         row = col;
19         // get the id of the class of the next event
20         col = eventClassIdentifiers.get(events[i].trim());
21         // now we have a row id that is the class of an event, and a column id that is
                the class of the following one
22         if (row != null && col != null) {
23            /* record this relation between these two events in the matrix,
24             * adding in the relative position the number of occurrences of this
                    relation */
25            relationMatrix[row][col] = relationMatrix[row][col] + trace.getCount();
26            // add the second event to the trace list
27            traceList.append(", " + eventClassAliases[col]);
28         } else { // if an error occurred, show it
29            error = true;
30            // stop the process
31            break;
32         }
33      }
34   } else { // if an error occurred, show it
35      error = true;
36   }
37 }
```

In questo modo, per ogni dipendenza tra due elementi all'interno di una traccia, vi sarà nella matrice alla posizione che identifica tale relazione, un valore pari alla somma di tutte le occorrenze di questa relazione all'interno del log.

Abbiamo la nostra matrice delle occorrenze delle relazioni tra le classi di eventi.

Partendo da essa, la footprint matrix [1, 3] da riportare nel report finale

di analisi è molto semplice.

La footprint matrix è una matrice N x N, dove per ogni relazione definita da una posizione nella matrice, viene inserito un simbolo che ne identifica un tipo specifico. Presa una coppia (A,B) nella matrice, dove A è un elemento riga e B un elemento colonna, le possibili relazioni sono le seguenti:

1. *Causality*: questo simbolo identifica una relazione da A a B, ovvero in altre parole che B dipende in maniera causale da A, e l'assenza della relazione inversa, ovvero da B ad A. Il suo simbolo è una freccia verso destra.

2. *Inverse Causality*: questo simbolo identifica una relazione da B a A, ovvero in altre parole che A dipende in maniera causale da B, e l'assenza della relazione inversa, ovvero da A ad B. Il suo simbolo è una freccia verso sinistra.

3. *Parallel*: questo simbolo identifica una relazione sia da A a B, sia da B ad A, ovvero che entrambi gli elementi dipendono dall'altro. Il suo simbolo è una doppia barra verticale (||).

4. *Choice*: questo simbolo identifica la totale assenza di relazione tra A e B, ovvero non esiste una dipendenza causale tra i due elementi. Il suo simbolo è il cancelletto (#).

Presa quindi la matrice delle occorrenze, l'algoritmo genera la footprint matrix nel seguente modo:

1. Il processo scorre la matrice delle occorrenze riga per colonna.

2. Fissando una riga, e quindi una classe di eventi, il processo scorre tutte le colonne, quindi tutte le classi di eventi, controllando il valore delle occorrenze in quella posizione.

3. Se la posizione controllata corrisponde alla diagonale della matrice, e quindi ad un confronto tra una classe di eventi e sé stessa, se il valore è maggiore di zero, allora viene segnato sulla footprint matrix una relazione di tipo Parallel, altrimenti se il valore è uguale a zero, segna l'assenza di relazione tra le due classi, ovvero Choice.

4. Se la posizione non è la diagonale, il processo controlla sia il valore di questa posizione, sia il suo inverso (quindi sia la posizione (A,B) che la posizione (B,A)).

5. Se i valori delle due posizioni sono entrambi maggiori di zero, la relazione segnata sulla footprint matrix sia nella posizione (A,B) che nella posizione (B,A) sarà Parallel, per quanto definito precedentemente.

6. Se invece entrambi i valori sono uguali a zero, la relazione sarà Choice in entrambe le posizioni all'interno della footprint matrix.

7. Se il valore della posizione (A,B) è maggiore di zero e quello della posizione (B,A) è uguale a zero, nella posizione (A,B) della footprint matrix la relazione sarà Causality, mentre nella posizione (B,A) sarà Inverse Causality.

8. In maniera opposta, se è il valore della posizione (B,A) ad essere maggiore di zero e quello della posizione (A,B) è uguale a zero, nella posi-

zione (A,B) della footprint matrix la relazione sarà Inverse Causality, mentre nella posizione (B,A) sarà Causality.

```
1 // for every event class in the matrix row line
2 for (int i=0; i<eventClassNumbers; i++) {
3   ...
4   // now calculate the relation between two events based on the values of the relation
        matrix
5   // for every event class in the matrix column line
6   for (int j=0; j<eventClassNumbers; j++)
7   {
8      String relation;
9      // get the value of i->j
10     int valueA = relationMatrix[i][j];
11     // if i and j are the same class (the relation matrix is NxN)
12     if (i == j) {
13        // if the value is lower than the barrier (that is 1)
14        if(valueA < barrier) {
15           // does not exist a relation between these two events
16           relation = choice;
17        } else {
18           // the relation is parallel
19           relation = parallel;
20        }
21     } else {// if they are not the same class
22        // we have to check the value of j->i
23        int valueB = relationMatrix[j][i];
24        // if the value A is higher than the barrier (that is 1)
25        if (valueA >= barrier) {
26           // if the value B is higher than the barrier (that is 1)
27           if(valueB >= barrier) {
28              // their relation is parallel
29              relation = parallel;
30           } else {
31              // there is a relation i->j
32              relation = casuality;
33           }
```

```
34          } else { // if instead the value A is lower than the barrier (that is 1)
35             // if the value B is higher than the barrier (that is 1)
36             if(valueB >= barrier) {
37                // there is a relation j->i
38                relation = inverseCasuality;
39             } else {
40                // does not exist a relation between these two events
41                relation = choice;
42             }
43          }
44       }
45       ...
46    }
47    ...
48 }
```

Dopodiché, conclusi i calcoli, il processo di analisi genera un documento XHTML per mostrare all'utente la lista delle tracce selezionate dai filtri ed utilizzate per l'analisi, la footprint matrix, la lista dei filtri utilizzati e la legenda degli elementi di righe e colonne della matrice (che come detto sono gli alias delle classi di eventi), riportando per ciascuno di essi i valori degli attributi che definiscono quella classe di eventi.

In Figura 5.10 viene mostrato un report di analisi generato a partire dal un log di eventi. Il report è stato poi esportato tramite l'apposita funzione di export che vedremo nella Sezione 5.3 per poter essere facilmente incluso in questo documento.

Footprint Matrix of exercise3

$\mathbf{L} = [\langle b, f, d, g \rangle^1, \langle a, c, e, g \rangle^1, \langle a, e, c, g \rangle^1, \langle b, d, f, g \rangle^1]$

	a	b	c	d	e	f	g
a	#	#	→	#	→	#	#
b	#	#	#	→	#	→	#
c	←	#	#	#	||	#	→
d	#	←	#	#	#	||	→
e	←	#	||	#	#	#	→
f	#	←	#	||	#	#	→
g	#	#	←	←	←	←	#

Filters

Selected Classifier: Event Name
Min Trace Occurences: 1

Matrix Legend

alias → concept:name
a → A
b → B
c → C
d → D
e → E
f → F
g → G

Figura 5.10: La footprint matrix ed il report di analisi

5.3 Import, export e stampa dei documenti

Il sistema come abbiamo visto gestisce e genera un discreto numero di documenti, tutti visualizzabili all'interno di due classi: `EventLogEditor` per i documenti testuali editabili (che siano log, documenti CSV o semplici note)

e `FootprintMatrixInfo` per i report in formato XHTML.

Per quanto riguarda il primo, abbiamo già visto come al suo interno possano essere aperti documenti testuali attraverso la funzione "Open" dell'interfaccia e salvati esternamente sempre in formato testuale attraverso quella di "Save As".

In questa sezione adesso prenderemo in esame due funzionalità dell'applicazione che permettono di aprire e salvare i log in formato diverso da quello testuale XML, nonché di aprire e salvare i report di analisi in formati diversi.

Alla fine vedremo la funzione di stampa dei documenti testuali, inserita in questa sezione in quanto attinente alla tematica dell'esportazione dei documenti al di fuori del sistema.

5.3.1 La funzione di import

La prima funzione che verrà mostrata è la funzione di import: questa funzione permette di caricare all'interno del sistema documenti CSV che seguono lo standard indicato nella Sezione 5.1.2, trasformandoli durante la fase di importazione in log XES ed aprirli all'interno di un `EventLogEditor` in formato XML, e documenti in formato HTML, aprendoli dopo una dovuta validazione all'interno di una finestra `FootprintMatrixInfo`.

Questa funzione è selezionabile dal menù principale, utilizzando il pulsante "Import". Una volta attivata, questa funzione mostrerà un finestra di selezione file come quella del pulsante "Open", con l'unica differenza che in questo caso saranno selezionabili solamente file di tipo CSV, TXT o HTML.

5.3.1.1 Da documenti CSV a log XES

Nel caso in cui l'utente selezioni un file di tipo CSV o TXT, la funzione procede con la lettura ed interpretazione del suo contenuto, con l'intento di elaborarlo e generare da esso un log XES in formato XML.

Il processo di generazione del log a partire da un documento in formato CSV è la stesso descritto nella Sezione 5.1.2, che questa volta viene utilizzato in automatico dal sistema all'import del documento. Non riporteremo qui di nuovo la stessa descrizione del processo, essendo identica, limitandoci a rimandare il lettore alla sezione dedicata.

5.3.1.2 I documenti HTML e XHTML

Il sistema permette di importare documenti HTML e XHTML all'interno di una finestra `FootprintMatrixInfo`, finestra che come abbiamo descritto è adibita proprio alla visualizzazione di documenti HTML/XHTML.

Il motivo di questa funzionalità è semplice e diverrà ancora più chiaro quando parleremo della funzione di export: essendo il contenuto di una finestra `FootprintMatrixInfo`, che di solito è un report di analisi, esportabile in formato XHTML in modo da permetterne il salvataggio, è utile avere anche una funzione di import per poter caricare nuovamente quel report nel sistema e visualizzarlo.

La funzione di import di documenti HTML/XHTML funziona non solo con i report di analisi precedentemente esportati, ma con qualsiasi documento di questo tipo che sia ben formattato e contenga il codice che lo definisce tutto all'interno dello stesso file (ovvero non deve avere referenze ad altri file

HTML dove risiede un altro blocco del suo codice).

Grazie a questa applicazione è quindi possibile aprire un qualsiasi documento HTML/XHTML ben formattato, visualizzarlo nel sistema ed infine esportarlo in altri formati messi a disposizione dall'applicazione.

Nel caso dunque in cui l'utente selezioni un file di tipo HTML dalla finestra di selezione, il sistema notificherà all'utente per prima cosa l'inizio del procedimento di validazione del documento, chiedendo se si voglia eseguire prima una validazione, oppure procedere al caricamento del documento senza validarlo.

La validazione è necessaria per accertare la corretta forma del documento e quindi una sua sicura corretta visualizzazione all'interno del sistema. Un documento non validato potrebbe essere contenere errori e quindi generare problemi in fase di visualizzazione.

Per contro però, la validazione richiede la connessione al sito del W3C[1] e può richiedere anche alcuni minuti prima di completare.

All'utente quindi viene data la possibilità di scegliere quale strada intraprendere.

In Figura 5.11 viene mostrato il pannello che notifica all'utente questa situazione.

Nel caso più interessante in cui l'utente decida di validare prima il documento (l'altro caso termina con un caricamento immediato del contenuto del documento all'interno del sistema), la funzione di import procederà come segue:

[1] *World Wide Web Consortium*

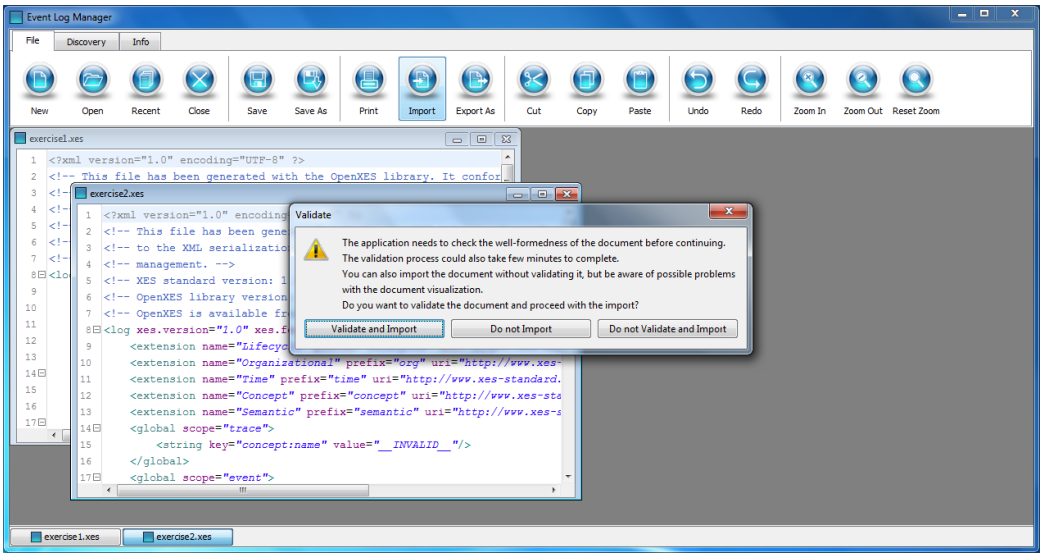

Figura 5.11: Import di un documento HTML

1. Controlla la presenza della connessione ad Internet, necessaria per la validazione.

2. In caso la connessione sia presente, recupera dall'URL specificato nel `DOCTYPE` del documento il file DTD[2] che ne definisce la struttura e lo utilizza per validare il documento. Se il documento risulta valido rispetto al DTD specificato, apre il suo contenuto all'interno di una finestra `FootprintMatrixInfo`, altrimenti restituisce un messaggio di errore che indica riga e colonna dell'elemento HTML che non rispetta la definizione del DTD specificato, in modo che possa essere corretto.

3. In caso in cui la connessione non sia presente, il sistema non può recuperare il DTD presso il sito del W3C e quindi non può accertare la validazione del documento; in questo caso notifica all'utente l'impossi-

[2] *Document Type Definition*

bilità di eseguire tale operazione e chiede se si voglia aprire il documento lo stesso (ed in caso affermativo lo apre come sopra), nonostante questo non sia stato controllato, tenendo presente la possibilità che vi siano errori in fase di visualizzazione se questo non era già ben formattato.

5.3.2 La funzione di export

Insieme alla funzione di import, ne viene fornita una opposta, che permette l'esportazione dei log XES in documenti esterni in formato CSV (Sez. 5.1.2) e dei report di analisi (ma in generale di qualsiasi documento HTML/X-HTML precedentemente aperto nel sistema) nei formati HTML/XHTML, PDF, LaTeX e PNG.

Questa funzione è attivabile dal pulsante "Export" dell'interfaccia e la sua esecuzione iniziale è simile alla funzione di "Save As": viene aperta una finestra di selezione dove l'utente può selezionare o creare il file di destinazione dell'esportazione, con l'unica differenza rispetto alla funzione di salvataggio dei tipi di file su cui poter esportare.

5.3.2.1 Esportare un log in formato CSV

Nel caso in cui l'utente selezioni un `EventLogEditor` con all'interno un log in formato XML e successivamente selezioni la funzione di Export, il sistema permetterà a lui la scelta del file di tipo CSV o TXT sul quale effettuare l'esportazione del log.

Successivamente, il sistema demanda il compito della conversione del log alla classe `XesXmlSerializer`, già vista nella Sezione 3.3.4.3.

Questa classe implementa l'interfaccia `XSerializer` delle librerie OpenXES per definire un serializzatore dal formato XML dei log XES, a quello CSV definito da questa applicazione.

Il procedimento è più semplice rispetto a quello inverso, eseguito dalla classe `CsvXesParser`, in quanto in questo caso il processo di validazione del log in formato XML che viene eseguito all'inizio dell'esportazione è demandato alla funzione vista nella Sezione 5.2.2, riutilizzando una operazione già implementata quindi.

A questa classe è quindi demandato solamente il compito di scorrere tutto il log (più precisamente la sua versione Java `XLog`) e di ricostruire passo dopo passo il relativo documento CSV:

1. Per prima cosa legge dal log tutte le chiavi degli attributi definiti per gli eventi e li utilizza per comporre e scrivere la prima riga del documento, quella contenente i nomi delle colonne.

2. Adesso, per ogni traccia del log, controlla se questa ha un attributo con chiave `identity:id` contenente l'identificatore della traccia; In caso affermativo, lo utilizzerà come valore della prima colonna di ciascuno dei suoi eventi, in caso contrario genera un identificatore appropriato secondo lo standard XES ed utilizza quest'ultimo.

3. A questo punto, per ogni evento della traccia in esame, crea una nuova riga nel documento, utilizzando come valore della prima colonna l'identificatore della traccia ed inserendo uno dopo l'altro i valori definiti da questo evento per gli attributi elencati nei campi colonna (lasciando

vuoti valori per i quali un dato attributo non è definito per l'evento in esame).

Al termine della conversione il documento viene scritto sul file di destinazione selezionato.

```
// write in the output the first column element
out.write("trace:id".getBytes());
// get a string of all the columns names
StringBuilder columnNameList = new StringBuilder();
for (String columnName : columnNames) {
   columnNameList.append(",").append(columnName);
}
// write in the output all the column names in one line
if (columnNameList.length() > 0) {
   out.write(columnNameList.toString().getBytes());
}
out.write(newLine.getBytes());
...
// for every trace of the log
for (XTrace trace : log) {
   // if the trace has attributes
   ...
   // for every event in the trace
   for (XEvent event : trace) {
      // if the event has attributes
      if (event.hasAttributes()) {
         // get them
         XAttributeMap attributeMap = event.getAttributes();
         // start recording in a single line the values for this event
         StringBuilder attributeValues = new StringBuilder();
         // first add the trace id of this event
         attributeValues.append(traceId);
         // for every column name in the first line
         for (String columnName : columnNames) {
            // add the CSV separator
            attributeValues.append(",");
```

```
32            // if the event attribute map contains the attribute defined by the column
                  name
33            if (attributeMap.containsKey(columnName)) {
34               // get it
35               XAttribute attribute = attributeMap.get(columnName);
36               String value = null;
37               // depending on the attribute type, read its value
38               if (attribute instanceof XAttributeLiteral || attribute instanceof
                     XAttributeDiscrete ||
39                  attribute instanceof XAttributeContinuous || attribute instanceof
                        XAttributeBoolean ||
40                  attribute instanceof XAttributeID) {
41                  value = attribute.toString();
42               } else if (attribute instanceof XAttributeTimestamp) {
43                  Date timestamp = ((XAttributeTimestamp)attribute).getValue();
44                  value = xsDateTimeConverter.format(timestamp);
45               } else {
46                  value = "___ERROR___";
47               }
48               // add the read value to the string line and go to the next column name
49               attributeValues.append(value);
50            }
51            /* if the event does not have this attribute, the process will go to the
                  next column
52             * leaving an empty value for this one (i.e "value1,value2,,value4", as CSV
                  standard) */
53         }
54         // write the just created line of values in the output and go to the next
               event
55         out.write(attributeValues.toString().getBytes());
56         out.write(newLine.getBytes());
57      }
58   }
59 }
```

5.3.2.2 Esportare un documento HTML/XHTML

Nel caso in cui invece l'utente selezioni una finestra contenente un documento HTML/XHTML e successivamente utilizzi la funzione di Export, il sistema proporrà lui attraverso la già vista finestra di selezione di creare o selezionare un file di uno dei seguenti tipi: HTML, PDF, LaTeX e PNG.

HTML

La selezione di questo tipo di file comporta una lettura del codice HTML/X-HTML della finestra (che all'utente ne mostra il risultato) ed una scrittura dello stesso sul file di destinazione, in maniera diretta, in modo da salvare questo contenuto per una successiva eventuale importazione.

```
1  try {
2     // set the busy state of the application
3     elmView.setBusyState(true);
4     Files.write(targetFile, xhtml.getBytes());
5     elmView.setBusyState(false);
6  } catch (IOException e) {
7     // error while writing the file
8     elmView.setBusyState(false);
9     getError = true;
10 }
11 // if we write the file correctly
12 if(!getError) {
13    // set the content as exported (this happens only with HTML export)
14    footprintMatrixInfo.setContentExported(true);
15    // get the absolute path of the file associated with this footprint matrix, if exist
16    String absolutePath = footprintMatrixInfo.getAbsolutePath();
17    // if the file path where we just exported is different from the last one, we update
           it
18    if (absolutePath == null) {
19       footprintMatrixInfo.setAbsolutePath(selectedDestinationFile);
```

```
20    } else if (!absolutePath.equalsIgnoreCase(selectedDestinationFile)) {
21       footprintMatrixInfo.setAbsolutePath(selectedDestinationFile);
22    }
23    // file exported correctly
24    fileExported = true;
25 } else {
26 ...
27 }
```

Questo tipo di export imposta la finestra della classe `FootprintMatrixInfo` come correttamente esportata, in modo che questa possa essere chiusa senza la necessità di conferma.

PDF

Questa conversione viene effettuata dalla libreria FlyingSaucer, descritta nella Sezione 2.2.4.

La libreria prende in input il contenuto della finestra, che deve essere un documento HTML o XHTML ben formato, e lo elabora generando un file PDF come output nella destinazione selezionata.

```
1  ITextRenderer renderer;
2  try {
3  // set the busy state
4  elmView.setBusyState(true);
5  // get the renderer
6  renderer = new ITextRenderer();
7  /* we set the renderer using the XHTML code generated by the AnalysisManager (so well-
       formed for sure),
8   * or the XHTML code imported from an external file (but checked during import, so well
       -formed too) */
9  renderer.setDocumentFromString(xhtml);
10 renderer.layout();
11 } finally {
```

```
12    elmView.setBusyState(false);
13 }
14 // try-with-resources Statement (close the resource automatically at the end in any
       case)
15 try (FileOutputStream fileStream = new FileOutputStream(selectedDestinationFile)) {
16       // set the busy state
17       elmView.setBusyState(true);
18       // ask the renderer to create the PDF in the target destination
19       renderer.createPDF(fileStream);
20       elmView.setBusyState(false);
21 } catch (IOException e) {
22 ...
23 }
```

LaTeX

Questa conversione viene effettuata dalla libreria HTMLtoLaTeX, descritta nella Sezione 2.2.4.

La libreria prende in input un file HTML (nel nostro caso dunque un file temporaneo viene generato con il contenuto della finestra, per poter essere così passato alla libreria) e restituisce nella destinazione selezionata un documento in formato LaTeX che ne rappresenta la corretta trasformazione.

```
1 // get the parser
2 Parser htmlLatexParser = new Parser();
3 try {
4    // set the busy state
5    elmView.setBusyState(true);
6    // parse the HTML temp file using the ParserHandler and get as result a LATEX file
         in the target path
7    URL latexConfig = this.getClass().getResource("/latex/config.xml"); // get the image
          from the URL
8    htmlLatexParser.parse(new File(htmlTempFile.toString()), new ParserHandler(
         targetFile.toFile(), latexConfig.toString()));
```

```
 9    elmView.setBusyState(false);
10 } catch (FatalErrorException e) {
11    // error while parsing the HTML temp file
12    elmView.setBusyState(false);
13    getError = true;
14    errorMessage = "Unable to parse the document and generate a LaTeX file.\n"
15               +"Please try again.";
16 }
```

PNG

Se infine l'utente seleziona di voler esportare il contenuto sotto forma di immagine, il sistema effettua una sorta di "ritaglio" della finestra e del suo contenuto, esattamente come mostrato al suo interno, e salva questo ritaglio nel file immagine selezionato.

```
 1 //get the frame dimension
 2 Dimension textAreaDimension = footprintMatrixTextArea.getSize();
 3 // get an image big as the frame dimension
 4 BufferedImage img = new BufferedImage(textAreaDimension.width, textAreaDimension.height
       , BufferedImage.TYPE_INT_ARGB);
 5 // access to the graphic
 6 Graphics imgGraphics = img.getGraphics();
 7 // paint the frame content into the image, disposing the content in that space
 8 // (the frame shows to the user a preview of how the content will be displayed inside
       the image)
 9 footprintMatrixTextArea.paint(imgGraphics);
10 // close the graphic
11 imgGraphics.dispose();
12 // now try to write the image object in the target file as a PNG
13 try {
14    // set the busy state
15    elmView.setBusyState(true);
16    // write the image object in the target file as a PNG
17    ImageIO.write(img, "png", new File(selectedDestinationFile));
```

```
18    elmView.setBusyState(false);
19 } catch (IOException e) {
20 ...
21 }
```

Il ritaglio della finestra significa proprio che il sistema prende il contenuto così come è mostrato dalla finestra e lo riporta su di una immagine.

Modificando infatti la dimensione della finestra `FootprintMatrixInfo`, il suo contenuto viene ridistribuito sulla nuova dimensione, essendo generato a partire da codice HTML.

Questa ridistribuzione porterà a ritagli diversi dello stesso contenuto, e quindi ad immagini più o meno grandi e con contenuto disposto diversamente.

All'utente quindi viene fornita la possibilità di modificare prima la dimensione della finestra, per adattare il contenuto come desidera, e poi effettuare l'export in formato PNG, ottenendo una immagine migliore e con una resa migliore.

Va comunque sottolineato che l'immagine sarà il ritaglio del contenuto visualizzato nella sua interezza, e non della parte visibile a schermo. Questo vuol dire che il contenuto ridistribuito in seguito al ridimensionamento apparirà come modificato, ma la parte di contenuto che nel sistema rimane in un punto non visibile della finestra (la parte accessibile tramite barre di scorrimento laterale) verrà comunque riportata nell'immagine.

5.4 Errori riscontrati sullo XES XSD 2.0 e sulla libreria OpenXES 2.0

Durante lo sviluppo di alcune funzioni dell'applicazione, in particolare utilizzando lo XSD Schema 2.0 di XES e le relative librerie OpenXES 2.0, abbiamo riscontrato alcuni errori nella loro implementazione.

5.4.1 Errore sullo XES Schema 2.0

Il più importante di tutti è stato quello relativo ad una particolare serie di definizioni nel file "xes2.0.xsd", che definisce lo Schema dello standard XES nella sua versione 2.0, dove di fatto queste definizioni erano in contrasto con la struttura dei log prodotti dalla libreria OpenXES 2.0 e quindi di conseguenza da tutte le applicazioni che ne facevano uso, tra tutte oltre a quella in esame, anche ProM[3].

In sostanza, un log prodotto dalle librerie ufficiali di XES, non era ritenuto corretto se validato rispetto allo Schema della stessa versione di XES.

Questo ovviamente creava una notevole incongruenza nel sistema, in quanto venivano generati log che non potevano essere validati e il sistema non riusciva a validare log esterni in quanto lo Schema li definiva non corretti nella loro forma prodotta attraverso le librerie Java di XES.

Questo problema, una volta scoperto, è stato riportato agli sviluppatori dello standard e della libreria e, dopo una serie di analisi incrociate sul codice e sullo Schema, quello che abbiamo ottenuto è stato l'aggiornamento dello Schema alla versione 2.2.

[3] http://www.promtools.org/doku.php

Questo nuovo Schema è stato ottenuto correggendo le definizioni incongruenti con lo standard presenti nello Schema 2.0, arrivando così ad un nuovo Schema conforme allo standard XES e adatto alla validazione dei log generati con le librerie OpenXES (che generavano invece già un log conforme alle specifiche dello standard e quindi non hanno avuto bisogno di modifiche).

Riportiamo un estratto della conversazione avuta con gli sviluppatori che mostra i problemi riscontrati nella versione 2.0 dello Schema ed il conseguente rilascio della versione 2.1 prima, e 2.2 poi, per la correzione di questi ultimi.

> OK, thanks, now I could reproduce your problem. Actually, I found several:
>
> The xmlns attribute in the log element is not supported (this is why you get the message you mention).
>
> Some attributes have a too-restricted type (xs:NCName, for example).
>
> The order of the child elements of the log is not adhered to, the log attributes should go first, not the extensions. This, I didn't know before, another thing learned...
>
> I've made an update of the schema's available: `http://www.xes-standard.org/xes21.xsd` and `http://www.xes-standard.org/xesext21.xsd`. These changes fixes problem 2., which leaves problems 1. and 3. For this, there is no easy fix in the schema, the attributes should simply go first, and the xmlns attribute should go. In case you're using (the Nightly build of) ProM, the Log package has also been updated to export the log by first exporting

> the attributes. So, importing the log into ProM and exporting it again should fix this problem (this will also remove the xmlns attribute for problem 1. from the log).
>
> . . .
>
> Another update. Based on your observations, I learned that the log attributes should now precede the extension elements etc. Clearly, this is not really desirable, as the standard prescribes that these attributes follow the classifiers and precede the traces. Therefore, I have changed the schema. There is now a new schema, see `http://www.xes-standard.org/xes22.xsd`. This one should support existing logs.

La versione 2.2 è attualmente disponibile online nel sito ufficiale dello standard XES ed utilizzata dall'applicazione sviluppata, tuttavia ancora deve essere rilasciata sotto forma di versione ufficiale, assieme alla relativa libreria OpenXES, in quanto i tempi per il rilascio di versioni ufficiali ha dei tempi ben definiti e abbastanza lunghi.

5.4.2 Errori sulla liberia OpenXES 2.0

Per quanto riguarda invece la libreria OpenXES 2.0, aggiungiamo al problema citato nella precedente sezione, altri due problemi riscontrati al suo interno: questi due errori, insieme ad un altro riscontrato all'interno della documentazione dello standard 2.0 riguardante il codice di un log, sono stati anch'essi segnalati e verranno modificati a partire dalla prossima versione.

Il primo è relativo alla classe `XExtensionManager`. Questa classe implementa un registro all'interno del quale sono presenti tutte le estensioni standard di XES, presentate nella Sezione 2.1.4.

Questo registro serve ad avere accesso diretto all'interno di una applicazione all'elenco di tutte queste estensioni registrate, comprese quelle personalizzate che si possono creare ed in seguito registrare a questa classe, in modo da renderle disponibili all'interno del sistema ovunque si voglia.

Abbiamo notato come questo registro contenga solo quattro delle sei estensioni standard di XES: l'estensione Cost e l'estensione ID, rappresentate rispettivamente dalla classe `XCostExtension` e dalla classe `XIdentityExtension`, non sono infatti presenti al suo interno.

Questo vuol dire che, richiamando il registro per ottenere le estensioni standard di XES da utilizzare per una data validazione, non si ottengono tutte quelle disponibili, con il conseguente problema di avere tutti gli attributi definiti da queste due estensioni non analizzabili.

La soluzione utilizzata all'interno del sistema per ovviare al problema è stata quella di inserire all'avvio del Controller queste due estensioni manualmente all'interno del registro, in modo da poter avere una lista completa a disposizione di tutte le funzioni dell'applicazione ed evitare spiacevoli inconvenienti di validazione.

Come accennato, in futuro verrà rilasciata una nuova versione della libreria che correggerà questo problema alla base, inserendo le due estensioni nel registro fin dall'inizio.

Il secondo problema riscontrato riguarda la classe `XesXmlSerializer`, classe che serve per passare da un log nel suo formato a oggetti (rappresentato

dalla classe `XLog` della libreria) alla rispettiva versione in formato testuale XML.

Questa classe, nel passaggio dal formato ad oggetti del log alla rappresentazione testuale, riporta come versione dello standard XES e della libreria OpenXES in tutti i documenti generati sempre la 1.0, anche se la versione di Schema e libreria utilizzati è la 2.0 ad esempio.

Analizzando il codice della classe, si nota che questa recupera questi dati da dei valori statici all'interno di una classe di utilità della libreria.

Probabilmente questa classe di utilità non è stata aggiornata nel passaggio della libreria dalla versione 1.0 alle successive e riporta ancora il vecchio valore sia per quanto riguarda lo standard associato che per quanto riguarda le librerie stesse.

Anche su questo un aggiornamento della libreria è stato messo in programma e porrà rimedio al problema.

Capitolo 6

SVILUPPO: GESTIONE DELL'APPLICAZIONE

In questo terzo ed ultimo capitolo sullo sviluppo dell'applicazione andremo a mostrare le funzionalità relative alla gestione della applicazione e delle sue componenti grafiche.

6.1 Interazione con l'interfaccia utente

Quando parliamo di interazione con l'interfaccia utente ci riferiamo a quelle azioni che l'utente può compiere sulla finestra principale per le quali quest'ultima deve implementare una opportuna soluzione.

Le azioni per le quali l'interfaccia ha la necessità di rispondere di conseguenza sono due:

1. la modifica delle dimensioni della finestra, tramite trascinamento dei bordi/angoli della stessa con conseguente ridimensionamento o aumento delle sue dimensioni, oppure tramite pulsante di ridimensionamento presente nella barra della finestra.

2. chiusura della finestra principale, tramite relativo pulsante nella barra della finestra oppure tramite altre funzioni di chiusura messe a disposizione dal sistema operativo (ad esempio su Microsoft Windows, la scorciatoia da tastiera ALT+F4, oppure tasto destro sull'icona nella barra delle applicazioni e selezione della voce "Chiudi finestra").

6.1.1 La modifica delle dimensioni della finestra principale

Per quanto riguarda la prima azione che l'utente può compiere, la finestra deve gestire la corretta visualizzazione dei pulsanti presenti nel menù principale e nella barra dei pulsanti in basso, e lo fa con due approcci differenti.

6.1.1.1 La gestione dei pulsanti nel menù principale

Per la gestione dei pulsanti del menù principale, viene utilizzato il layout manager `MenuAreaTabLayout`: questo manager viene utilizzato dalla finestra principale per la disposizione dei pulsanti all'interno del menù in alto.

Quando la finestra è abbastanza grande affinché tutti i pulsanti possano essere mostrati, questo manager non fa altro che disporli in sequenza, nell'ordine prestabilito.

Quando invece lo spazio a disposizione è minore di quello necessario, il `MenuAreaTabLayout` procede nel modo seguente:

1. Calcola quanti pulsanti possono entrare adesso nel menù, considerando la nuova dimensione disponibile.

2. Se *n* è il numero dei pulsanti così determinato, inserisce i primi *n-1* dell'elenco nel menù.

3. Nell'ultimo posto disponibile per i pulsanti, inserisce un pulsante denominato "More"; Questo è un pulsante extra che, se selezionato, apre un pop-up sotto di sé, sempre posizionato nell'area visibile dello schermo.

4. Inserisce tutti i pulsanti rimasti all'interno del pop-up del pulsante More.

Così facendo otteniamo per ogni dimensione, un nuovo menù dei pulsanti definito da *n-1* elementi del menù ed un pulsante speciale che racchiude i rimanenti in un menù secondario, apribile su richiesta, che prende una posizione sempre visibile nello schermo.

In Figura 6.1 è mostrato il caso in cui la dimensione della finestra sia stata diminuita ed il pulsante more si renda necessario.

Si può arrivare fino ad avere un menù composto solo dal pulsante More e tutti i pulsanti del sistema all'interno del suo sotto-menù.

```
1 private int howManyInOneRow() {
2    // get the insets of the container and the hgap
3    Insets containerInsets = targetPanel.getInsets();
4    int horizGap = getHgap();
5    // get the width of the container without the insets and hgap
```

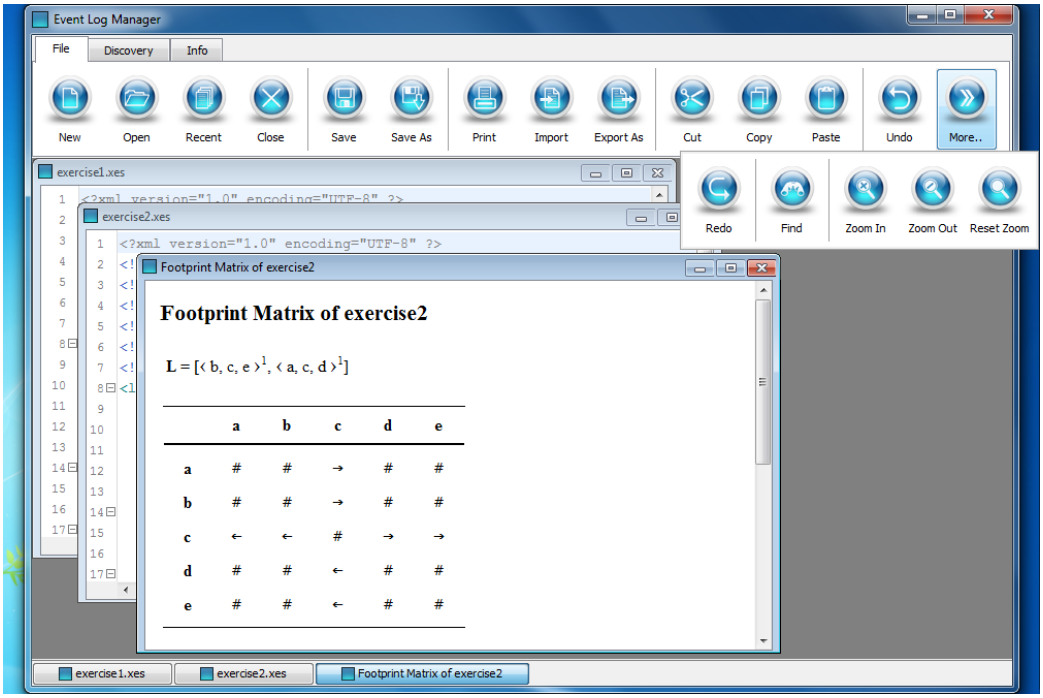

Figura 6.1: Cosa accade al menù se la finestra viene ridimensionata

```
6    int containerWidth = targetPanel.getWidth() - (containerInsets.left +
         containerInsets.right + horizGap*2);
7    int rowWidth = 0;
8    // for every component of the container
9    for (int i = 0; i < targetPanelComponents.length; i++) {
10      // get the component
11      Component component = targetPanelComponents[i];
12      // if the component has to be displayed
13      if (component.isVisible()) {
14         // get the size of the component
15         Dimension componentSize = component.getPreferredSize();
16         // if there is no space on the first row for displaying this component, return
               the index of it (that is also the number of component before it that can
              be displayed)
17         if (rowWidth + componentSize.width > containerWidth) {
18            return i;
19         }
```

```
20          // otherwise add the component size and the hgap to the total amount and go on
                checking
21          rowWidth = rowWidth + componentSize.width + horizGap;
22        }
23    }
24    // if all the components can fit the first row, return the total number
25    return targetPanelComponents.length;
26 }
```

6.1.1.2 La gestione dei pulsanti nella barra in basso

Per quanto invece riguarda la gestione dei pulsanti presenti nella barra in fondo alla finestra principale, il layout manager `OneRowFlowLayout` è quello adibito a questo compito.

Il suo obiettivo è far si che, indipendentemente dal numero di pulsanti presenti nella barra, tutti siano visualizzati e risiedano in un'unica riga.

Questo viene ottenuto nel modo seguente:

1. Per ogni `EventLogEditor` che viene creato, un pulsante `FooterAreaButton` viene inserito nella barra in fondo alla finestra principale. Se lo spazio a disposizione nella barra è maggiore della dimensione del pulsante necessaria affinché il nome dello stesso (che ricordiamo è lo stesso della finestra che gestisce) venga mostrato per esteso, il pulsante viene aggiunto alla barra utilizzando la dimensione che esso richiede.

2. Se invece nella barra non c'è spazio sufficiente per un pulsante di tale dimensione, il manager calcola la dimensione massima che un pulsante possa avere affinché tutti i pulsanti che devono essere visualizzati riescano ad entrare nella barra (divide in sostanza la dimensione della

barra per il numero dei pulsanti da visualizzare, tenendo conto dei vari margini laterali ed intermedi, ottenendo così la dimensione massima per un pulsante); a questo punto controlla ogni pulsante, se la dimensione da esso richiesta è minore della massima consentita, lo lascia inalterato, altrimenti lo ridimensiona in lunghezza alla massima consentita (quindi tagliando la parte finale del nome del bottone ed inserendo dei puntini di sospensione).

```
1 private Dimension selectSize(Dimension wantedSize, Dimension maxPossibleSize) {
2    int width = 0;
3    int height = 0;
4    // if the width of the wanted size is littler, take that, otherwise use the max
         possible size width
5    if (wantedSize.width < maxPossibleSize.width) {
6       width = wantedSize.width;
7    } else {
8       width = maxPossibleSize.width;
9    }
10   // if the height of the wanted size is littler, take that, otherwise use the max
         possible size height
11   if (wantedSize.height < maxPossibleSize.height) {
12      height = wantedSize.height;
13   } else {
14      height = maxPossibleSize.height;
15   }
16   // return a new dimension for the component built with the smallest width and height
17   return new Dimension(width, height);
18 }
```

Così facendo i pulsanti più grandi vengono ridimensionati per fare spazio all'ultimo arrivato, fino a quando tutti i pulsanti non hanno la stessa dimensione (ovvero quando ogni pulsante risulta più grande rispetto alla dimensione massima consentita, e quindi tutti vengono ridimensionati in base

a quest'ultima). A questo punto un ulteriore pulsante inserito ridimensionerà tutti i pulsanti presenti in maniera proporzionale, rendendoli sempre più piccoli in lunghezza, ma tutti visibili e utilizzabili per la gestione delle finestre.

Questa gestione dei pulsanti nella barra finale, avviene sia per via di un ridimensionamento della finestra principale (e quindi in conseguenza della diminuzione dello spazio disponibile per la visualizzazione dei pulsanti già presenti nella barra), sia per via di un aggiunta di un ulteriore pulsante alla barra quando questa è già piena e non ha spazio sufficiente per visualizzarlo (lo spazio a disposizione quindi non è diminuito per via di un ridimensionamento della finestra, ma perché ci sono troppi pulsanti rispetto alla disponibilità).

Se prendiamo ad esempio la Figura 6.1 e la Figura 6.3 possiamo fare alcuni paragoni: mentre nella prima, nonostante la finestra sia stata ridimensionata (lo si può notare dall'apparizione del pulsante More nel menù principale), lo spazio per la visualizzazione dei pulsanti nella barra a fine finestra ancora è sufficiente e quindi questi non sono stati ridimensionati, nella seconda questi ultimi sono stati ridimensionati nonostante la finestra sia ancora alle sue dimensioni originali: questo accade semplicemente perché il numero di pulsanti era più elevato dello spazio a disposizione per visualizzarli alla loro dimensione desiderata ed è stato necessario un ridimensionamento complessivo.

6.1.2 La chiusura della finestra principale

Adesso vedremo come viene gestita la seconda azione che l'utente può compiere sulla finestra principale che richiede una gestione particolare da parte di quest'ultima.

Quando un utente decide di voler chiudere l'applicazione, il listener preposto all'ascolto e gestione della finestra, tale `MainFrameWindowListener`, per prima cosa chiede una conferma, per accertarsi che l'utente voglia davvero chiudere il sistema e i documenti aperti in esso e non sia un semplice errore di selezione dei pulsanti.

Una vola accertato che non si tratta di un errore, il sistema, prima di terminare, controlla se ci sono documenti ancora aperti, ed agisce come segue:

1. Se non sono presenti documenti aperti, il sistema termina.

2. Se sono presenti documenti aperti, il sistema inizia a chiuderli uno alla volta, richiamando su di essi la funzione Close definita precedentemente.

```
1 public void windowClosing(WindowEvent e) {
2    // retrieve the user's choice
3    String message = "Do you really want to close the application?";
4    String title = "Close";
5    Object[] options = {"Yes", " No "};
6    int choice = elmView.showOptionDialog(message, title, JOptionPane.QUESTION_MESSAGE,
         options, options[0]);
7    // check user's decision
8    if (choice == 0) { // if yes
9       boolean savingFlag = true;
10      // get all opened internal frames
11      JInternalFrame[] openedInternalFrames = elmView.getContentArea().getAllFrames();
12      // for every internal frame
```

```
13      for (JInternalFrame internalFrame: openedInternalFrames) {
14         try {
15            // set the internal frame that we are going to close as selected
16            internalFrame.setSelected(true);
17         } catch (PropertyVetoException ex) {} // we do not need to do anything if an
               exception has been thrown
18         // if we selected correctly the internal frame to close
19         if (internalFrame.isSelected()){
20            // call the close operation and get the result
21            savingFlag = closeInternalFrame(internalFrame);
22         } else { // we were not able to set the frame as selected for some reason
23            // set the result of the process as error
24            savingFlag = false;
25            elmView.showErrorMessage("Unable to close the document. Please try again.");
26         }
27         // if we did not close the internal frame successfully
28         if (!savingFlag) {
29            // stop the closing process
30            break;
31         }
32         // otherwise go to the next internal frame
33      }
34      // at the end of the cycle, we check if we successfully close all the internal
            frames or not
35      if (savingFlag) {
36         // set the close operation of the window as exit
37         elmView.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
38      } else {
39         // set the close operation of the window as do not close
40         elmView.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
41      }
42   } else { //if no or the user close the dialog
43      elmView.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
44   }
45 }
```

In Figura 6.2 si può vedere la richiesta di conferma da parte del sistema

alla chiusura dell'applicazione, mentre avevamo già visto in Figura 4.7 i messaggi di richiesta di salvataggio durante una chiusura di finestra (che anche in questo caso vengono riproposti).

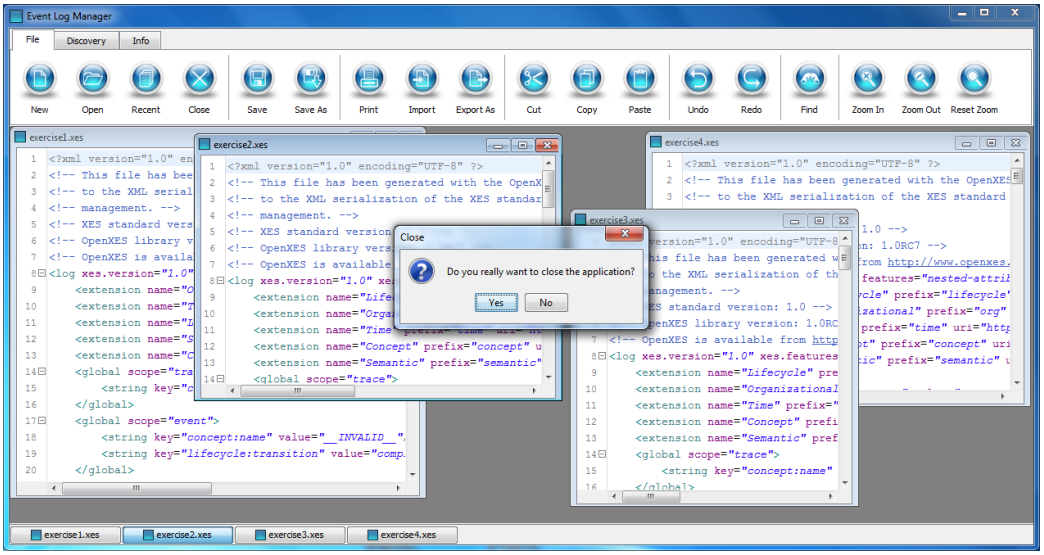

Figura 6.2: La gestione della chiusura dell'applicazione

Come abbiamo visto nella Sezione 4.7, questa funzione dapprima controlla se il documento risulta salvato (o esportato, a seconda dei casi) e se il file esiste ancora; se la risposta è positiva, la funzione chiude la finestra, se è negativa invece chiede all'utente se desidera salvare di nuovo il documento prima di chiuderlo oppure chiuderlo senza salvare.

In questo modo, per ogni documento aperto, la funzione Close verifica il suo stato e, se necessario, prima di chiuderlo chiede all'utente se desidera o meno salvarlo.

Dopo aver processato un documento e concluso le operazioni con una chiusura, passa al successivo, fino a quando tutti i documenti non risultano chiusi e quindi l'applicazione può terminare senza perdita di dati.

É importante notare che, nel caso una delle funzioni di chiusura di un documento dovesse non andare a buon fine (ad esempio quando il documento non è salvato, l'utente dapprima sceglie di salvare il file, ma poi annulla la procedura di salvataggio), il documento processato rimarrà aperto (data la scelta dell'utente) e il sistema non si arresterà, rimanendo ancora attivo a disposizione dell'utente.

6.2 Disposizione di una nuova finestra all'interno dell'applicazione

Quando il sistema inserisce una nuova finestra interna, la posizione nella quale viene posta all'interno della applicazione non è causale, ma frutto dell'elaborazione della classe `CustomDesktopManager`.

Questo manager ha il compito di trovare una posizione libera per la nuova finestra, in modo che non si sovrapponga alle altre già presenti (se questo è possibile, in base allo spazio a disposizione rimasto).

Il manager disegna una ipotetica diagonale dall'angolo in alto a sinistra all'angolo in basso a destra, e la utilizza per posizionare le varie finestre ad una distanza prestabilita le une dalle altre in questa diagonale, in uno stile chiamato "a cascata".

Così facendo, posiziona la prima finestra aperta nell'angolo in alto a sinistra, la seconda poco sotto di questa con l'angolo in alto a sinistra della finestra sempre sulla diagonale definita, e così via a scendere fin quando c'è spazio a disposizione.

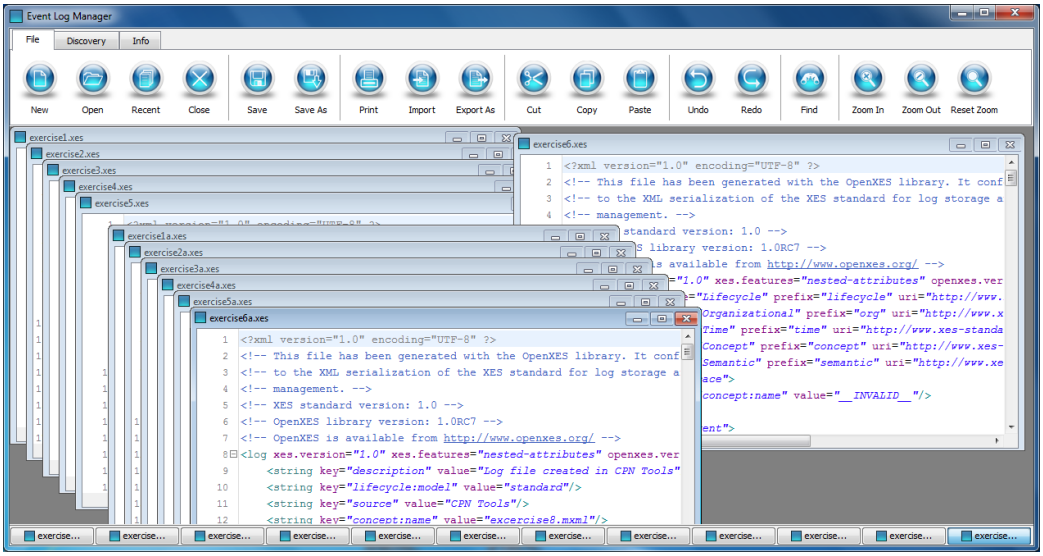

Figura 6.3: Il sistema a cascata per il posizionamento delle finestre

Nel caso lo spazio nella diagonale sia terminato, ovvero la prossima finestra dovrebbe avere la propria barra al di sotto della visibilità della schermata, il manager inizia a disporre le successive finestre al centro dell'area, una sopra l'altra.

Questo caso è l'ultima soluzione presa in considerazione dal manager nell'eventualità in cui non ci sia più modo di disporre le finestre in maniera di non sovrapposizione totale, ma alla quale si arriva davvero di rado, solo come ultima risorsa.

Infatti non solo il numero di finestre che possono giacere sulla diagonale è molto alto e quindi raramente capita di avere così tante finestre aperte tutte assieme, ma inoltre va tenuto conto di una funzione speciale del `CustomDesktopManager`: il riutilizzo dello spazio.

Capita spesso che una finestra aperta venga poi chiusa oppure spostata dalla sua posizione iniziale, lasciando libero uno spazio sulla diagonale. Que-

sti spazi vuoti che si vengono a creare vengono riutilizzati dal manager per il posizionamento delle future finestre.

In Figura 6.3 si può vedere come il sistema posizioni le finestre al loro inserimento nell'applicazione a cascata lungo una diagonale. Nell'esempio la finestra "exercise6.xes" è stata spostata dalla sua posizione iniziale, così facendo ha lasciato uno spazio vuoto nella diagonale che verrà riutilizzato dal manager: la prossima finestra aperta verrà posizionata in quello spazio vuoto e non in fondo alla cascata di finestre.

```
1  public void positionFrame(JInternalFrame f) {
2     JInternalFrame targetInternalFrame = f;
3     // if the position of this component is 0,0 it means that this component doesn't
          have a position yet on the screen
4     if ((targetInternalFrame.getX() == 0) && (targetInternalFrame.getY() == 0)) {
5        // get all the components (Internal Frames) in the container (JDesktop Pane)
6        JInternalFrame[] allInternalFrames = null;
7        synchronized (targetInternalFrame.getDesktopPane().getTreeLock()) {
8           allInternalFrames = targetInternalFrame.getDesktopPane().getAllFrames();
9        }
10       // starting point where the research begins
11       Point startingPoint = new Point(0,0);
12       // size of the container (JDesktop Pane) where to research free space
13       Dimension containerSize = targetInternalFrame.getDesktopPane().getSize();
14       // point where to place the component at the end
15       Point position = null;
16
17       // try to find the point
18       position = firstFreePosition(startingPoint, containerSize, allInternalFrames,
              targetInternalFrame);
19       // if we found a good position, set the location of the component
20       if (position != null) {
21          targetInternalFrame.setLocation(position.x,position.y);
22       }
23       /* if we didn't find a point, the component get positioned
```

```
24      * by the application in a default position on the screen */
25    }
26 }
27 ...
28
29 private Point firstFreePosition(Point startingPoint, Dimension containerSize,
       JInternalFrame[] allInternalFrames, JInternalFrame targetInternalFrame) {
30    // the space is free?
31    boolean isFree = false;
32    // point that represents at the end of the process the first free position found,
          starting from the starting point
33    Point freePosition = new Point(startingPoint);
34    // offset to use when searching a new free position, when the last checked was
          occupied
35    int offset = 20;
36
37    // until we don't find a free space, continue to search (we exit from this loop when
           we find a free space or with an inside return statement in case we finish the
          space where to search)
38    while (!isFree) {
39       // we suppose to find a free space at the actual value of the variable
             freePosition and we set to true the value of isFree
40       isFree = true;
41       // then we check if our hypothesis is right, looking if there is already an
             internal frame in that position or not
42       for (int i = 0; i < allInternalFrames.length; i++) { //for every component in the
              container
43          // if the internal frame that we are checking is the same that we try to
                position, skip to the next one
44          if (allInternalFrames[i].equals(targetInternalFrame)) continue;
45          // otherwise, get the position of the internal frame that we are checking now
46          Rectangle checkedPosition = allInternalFrames[i].getBounds();
47          // check if the freePosition that I suppose to be free, is instead occupied by
                this internal frame
48          if (checkedPosition.x == freePosition.x && checkedPosition.y == freePosition.y
                ) {
49             // if this internal frame is in the position that we supposed empty, then we
```

```
                need to check a new position
50              freePosition.x = freePosition.x + offset;
51              freePosition.y = freePosition.y + offset;
52              // if the new position that we want to check as next is too close to the
                    borders of the main window
53              if ((freePosition.x + offset) > containerSize.width || (freePosition.y +
                    offset) > containerSize.height) {
54                 return null; // it is not possible to find an empty position for this
                       internal frame
55              }
56              // we were wrong at the beginning on our supposition, this position is not
                    empty, so we set isFree as false
57              isFree = false;
58              // we don't need to check the remaining internal frames, because we already
                    found one in the position we were looking for
59              break;
60          }
61       }
62    }
63    // if no internal frames have been found in the freePosition we were looking for,
          then isFree has remained true and we have found an empty position
64    return freePosition;
65 }
```

6.3 Gestione di molteplici finestre: i FooterAreaButton

Durante l'utilizzo dell'applicazione, può capitare di avere spesso aperte più finestre di editing, per le quali magari abbiamo generato anche qualche report di analisi, ottenendo come risultato un notevole numero di finestre all'interno dell'interfaccia grafica.

Mentre il compito della loro disposizione al primo inserimento delle stesse nella finestra principale è compito della classe `CustomDesktopManager`, che vedremo in dettaglio nella Sezione 6.1, una volta che queste sono inserite possono essere gestite attraverso la barra dei pulsanti che si trova in fondo alla finestra principale.

Questa barra conterrà una serie di pulsanti `FooterAreaButton`, uno per ciascuna finestra aperta, sia che si tratti di `EventLogEditor`, sia di `FootprintMatrixInfo`, che vedremo meglio nella Sezione 5.2.

Ogni pulsante è collegato tramite un identificatore alla finestra ad esso associata: questo identificatore permette ai listener `FooterAreaButtonActionListener` (per la gestione del pulsante) e `PopupMenuItemActionListener` (per la gestione degli elementi del suo pop-up) di identificare a quale finestra applicare le modifiche dettate dall'utilizzo del suo pulsante.

Le azioni possibili sui pulsanti di questa barra sono due, il click su uno di essi con il tasto sinistro del mouse ed il click con il tasto destro.

In corrispondenza del click sinistro su un pulsante, queste sono le possibili conseguenze:

1. Se la finestra associata al pulsante non è selezionata (quindi non è in primo piano rispetto alle altre), viene selezionata (ed appare quindi sopra le altre aperte).

2. Se la finestra associata al pulsante è quella selezionata, viene minimizzata, ovvero "scompare" dalla finestra principale, liberando spazio per le altre.

3. Se la finestra associata al pulsante è attualmente minimizzata, questa viene riportata nella finestra principale in primo piano rispetto alle altre, ovvero viene visualizzata e selezionata.

```
1 if (internalFrame.isIcon()) { // if the internal frame is minimised
2    try {
3       // set the internal frame maximised
4       internalFrame.setIcon(false);
5    } catch (PropertyVetoException exception) {
6       elmView.showErrorMessage("Unable to maximize the window. Please try again.");
7    }
8 } else if(internalFrame.isSelected()) { // if the internal frame is maximised and
      selected
9    try {
10      // set the internal frame minimised
11      internalFrame.setIcon(true);
12   } catch (PropertyVetoException exception) {
13      elmView.showErrorMessage("Unable to minimize the window. Please try again.");
14   }
15 } else { // if the internal frame is maximised but not selected
16   try {
17      // set the internal frame selected
18      internalFrame.setSelected(true);
19   } catch (PropertyVetoException exception) {
20      elmView.showErrorMessage("Unable to select the window. Please try again.");
21   }
22 }
```

In corrispondenza invece del click destro su un pulsante, quello che si otterrà è l'apertura di un pop-up su di esso, con la voce "Close" all'interno.

Selezionando questa voce, si invocherà la funzione di chiusura della finestra, vista nella Sezione 4.7, che effettuerà gli opportuni controlli sullo stato del file (salvato/esportato correttamente o meno) e poi procederà alla chiusura della finestra, operando di fatto in maniera identica alla scelta del

pulsante Close del menù principale o del pulsante di chiusura nella barra della finestra.

```
1 String [] parts = e.getActionCommand().split("-", 2);
2 String command = parts[0];
3 String internalFrameID = parts[1];
4
5 switch (command) {
6    // if the command is to close the target internal frame
7    case "close":
8            JInternalFrame internalFrame = findJInternalFrame(internalFrameID);
9            // if we have found the associated internal frame, close it
10           if (internalFrame != null) {
11              closeInternalFrame(internalFrame);
12           } else {
13              elmView.showErrorMessage("There is not any document associated with this
                     button.");
14           }
15           break;
16   default: break;
17 }
```

6.4 Gestione delle configurazioni

Selezionando il pulsante "Settings" del menù Info dell'applicazione, quella che ci verrà mostrata sarà la schermata in Figura 6.4.

Attraverso questo pannello è possibile modificare le configurazioni del sistema, sia per quanto riguarda lo standard XES (scheda "XES" del pannello), sia per quanto riguarda alcune funzioni dell'applicazione (scheda "General" del pannello).

I valori di questi campi sono memorizzati in file di configurazione testuali e gestiti dal modello, che si occupa di recuperare i dati al loro interno e

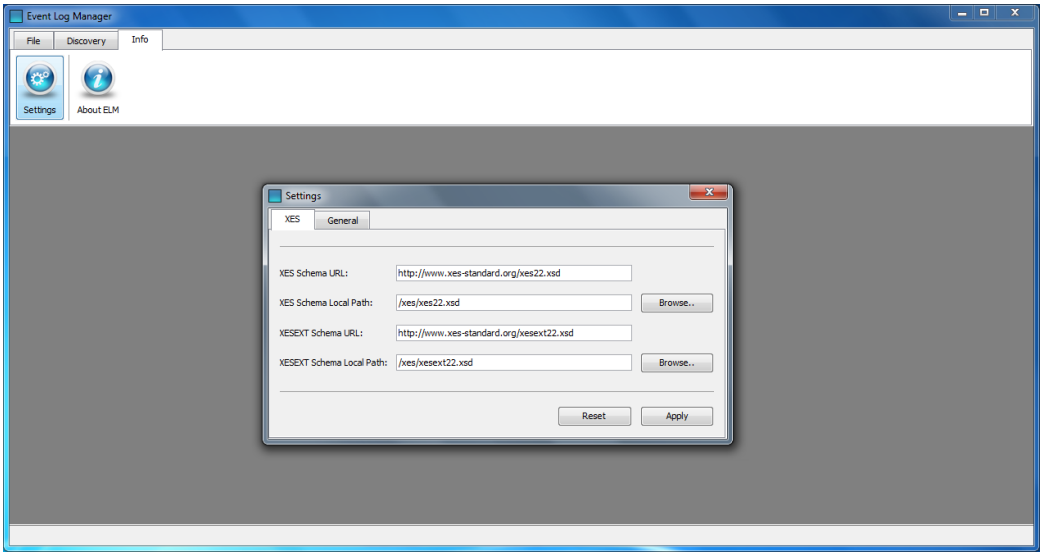

Figura 6.4: Il pannello delle impostazioni: la scheda XES

modificarli.

Così attraverso la scheda XES, sarà possibile modificare i riferimenti sia online che locali allo Schema di XES e e quello delle estensioni XESEXT.

Attraverso la scheda General, mostrata in Figura 6.5, sarà invece possibile modificare il numero di documenti da elencare nel menù a tendina del pulsante Recent (menù File) e la distanza in secondi che devono avere gli eventi quando questi vengono generati dalle apposite funzioni viste nella Sezione 5.1.3.

6.5 Note sul codice: lingua, commenti e JavaDoc

In questa ultima sezione riportiamo alcune note riguardanti il lavoro svolto.

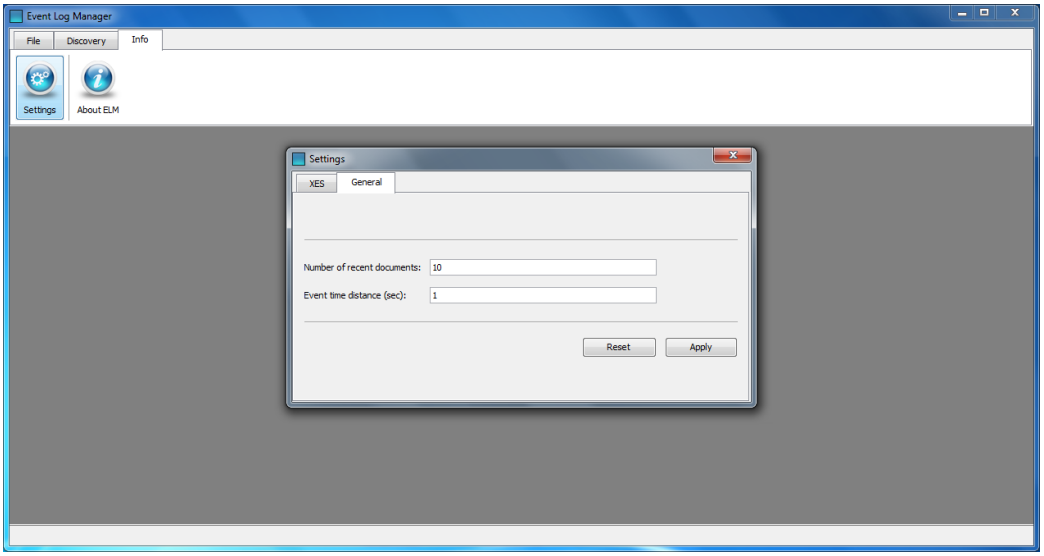

Figura 6.5: Il pannello delle impostazioni: la scheda General

Durante lo sviluppo dell'applicazione, al momento di scegliere la lingua da utilizzare per l'interfaccia utente, le componenti del codice Java, i commenti e non ultima la documentazione allegata al progetto sotto forma di JavaDoc, la scelta è ricaduta sulla lingua inglese.

La scelta pare ovvia se consideriamo l'intento di rendere questa applicazione utilizzabile da un numero più vasto possibile di utenti, sia in campo accademico da docenti e studenti, sia in ambito professionale da aziende e professionisti del settore.

La lingua inglese è ormai da anni la lingua ufficiale dell'Unione Europea, nonché la più parlata al mondo. Questa scelta quindi è sembrata la migliore per rendere l'applicazione non solo facilmente utilizzabile da una moltitudine di utenti di diversi paesi, ma anche estendibile e migliorabile da parte di sviluppatori non solo italiani, ma anche di qualsiasi Università o azienda estera.

Questa applicazione infatti è costituita da un codice che utilizza variabili, metodi e classi con nomi in inglese di facile comprensibilità, commenti dettagliati al codice che spiegano le operazioni svolte e una documentazione JavaDoc precisa ed esplicativa delle classi e di tutti i loro metodi, tutti in lingua inglese.

L'obiettivo come detto era quello di ottenere la massima comprensione del codice scritto per questo lavoro e la possibilità di avere una facile manutenzione ed estensione dello stesso in futuro.

Capitolo 7

CASI D'USO

In questo capitolo verranno mostrati due casi d'uso dell'applicazione, ciascuno riportante le operazioni principali che si possono eseguire per quanto riguarda rispettivamente la generazione e l'analisi di un log.

7.1 La generazione di un log

Per quanto riguarda la prima funzionalità, quella di generazione, una delle prime operazioni che l'utente può fare è quella di selezionare il pulsante "New" nel menù File dell'interfaccia, e successivamente selezionare l'opzione di apertura di un nuovo editor.

In Figura 7.1 viene mostrata l'esecuzione scelta richiesta all'utente riguardo al tipo di generazione che si desidera eseguire.

A questo punto l'utente all'interno del nuovo editor può iniziare a scrivere il codice del log che intende generare, utilizzando due metodi:

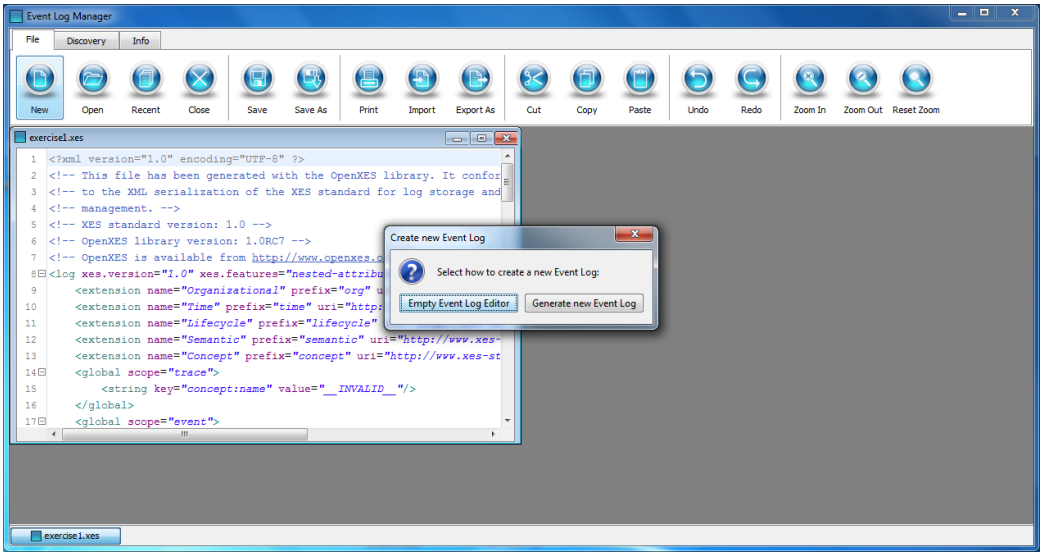

Figura 7.1: La scelta del metodo di generazione di un nuovo log

1. Scrivendo direttamente il codice del log, supportato dalle funzioni Template e Completion viste nella Sezione 5.1.1.

2. Definendo il log attraverso la semantica in formato CSV vista nella Sezione 5.1.2.

Nel primo caso, l'utente scrive l'identificatore del template che vuole caricare nell'editor, e premendo successivamente CTRL + SHIFT + SPAZIO, carica al posto dell'identificatore il blocco di codice ad esso associato.

In Figura 7.2 è mostrato un esempio composito del richiamo di alcuni template all'interno dell'editor.

A questo punto l'utente può completare i template caricati con i dati mancanti, utilizzando la funzione Completion: posizionandosi su un campo da compilare e premento CTRL + SPAZIO, si aprirà un menù a tendina come

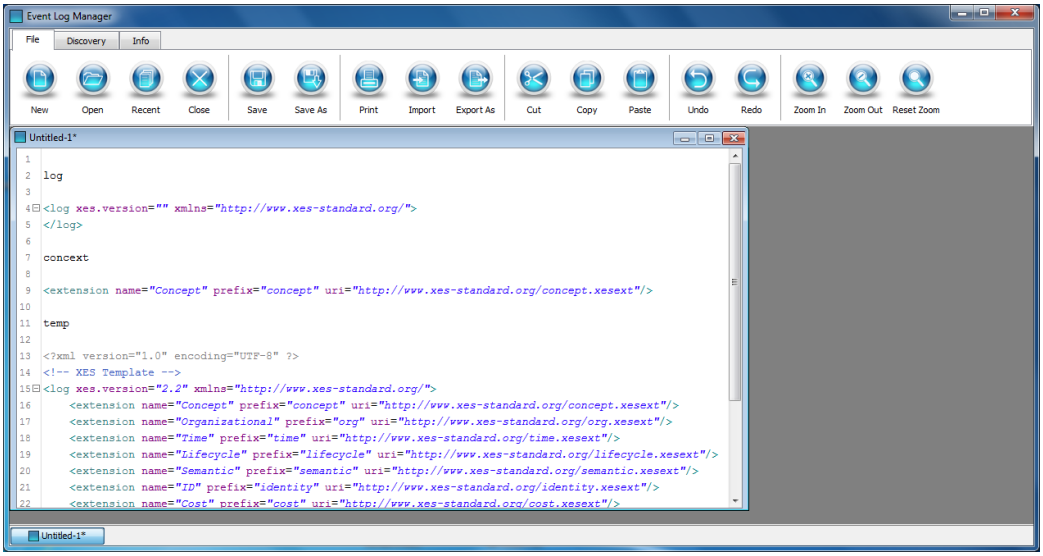

Figura 7.2: Generazione di un log utilizzando i `template`

quello mostrato in Figura 7.3, attraverso il quale l'utente potrà selezionare la parola chiave da inserire.

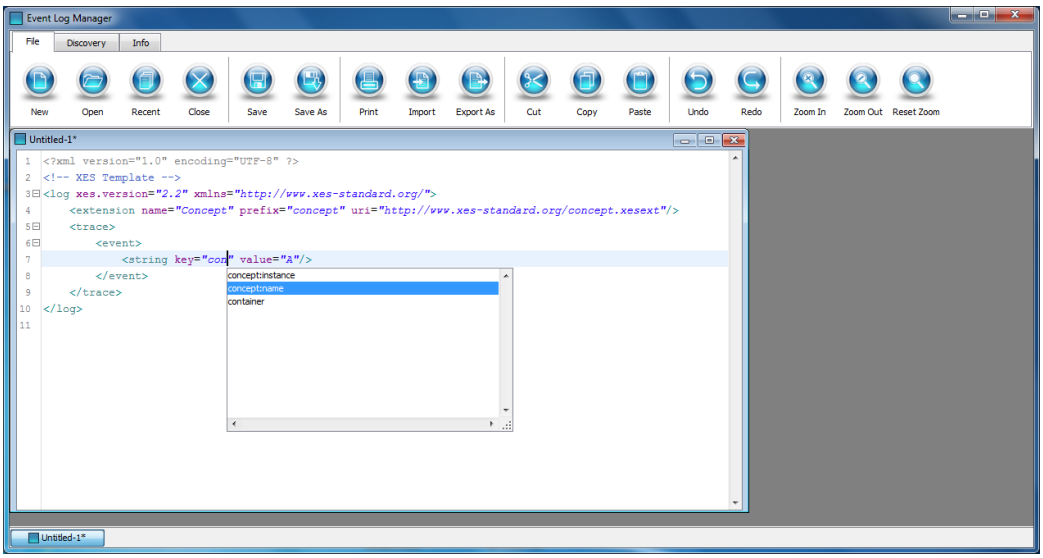

Figura 7.3: Il menù a tendina delle possibili parole chiave

L'utente può anche iniziare a scrivere una parola e successivamente uti-

lizzare questa funzione, in modo da restringere il campo delle possibilità e visionare nel menù a tendina solo il sotto-insieme delle parole chiave che inizia con la parte di testo già digitata.

Nel secondo caso invece, l'utente prima definisce il log all'interno dell'editor, come mostrato nell'editor a sinistra in Figura 7.4, dopodiché utilizza il pulsante "Generate" nel menù Discovery per processare tale contenuto ed ottenere in output su un nuovo editor il log in formato XML, mostrato sempre nella stessa Figura 7.4, questa volta a destra.

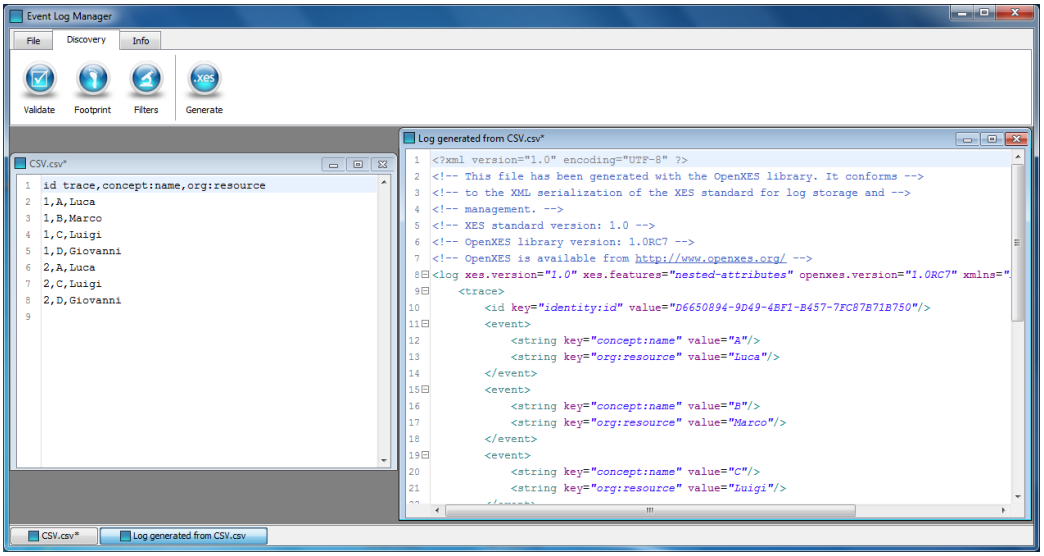

Figura 7.4: La generazione di un log a partire dalla semantica CSV

Per la definizione delle colonne del documento CSV, l'utente può farsi aiutare dalla funzione Completion descritta precedentemente.

Riprendendo la Figura 7.1, possiamo notare come l'utente possa utilizzare da questo pannello anche un altra funzione per la generazione di un log. Selezionando infatti il pulsante per la generazione di un nuovo log, l'utente troverà disponibile il pannello in Figura 7.5. All'interno di questo pannello

potrà utilizzare il linguaggio di espressioni descritto nella Sezione 5.1.3 per definire l'insieme delle possibili tracce del log, a partire dalle sequenze di attività che le compongono.

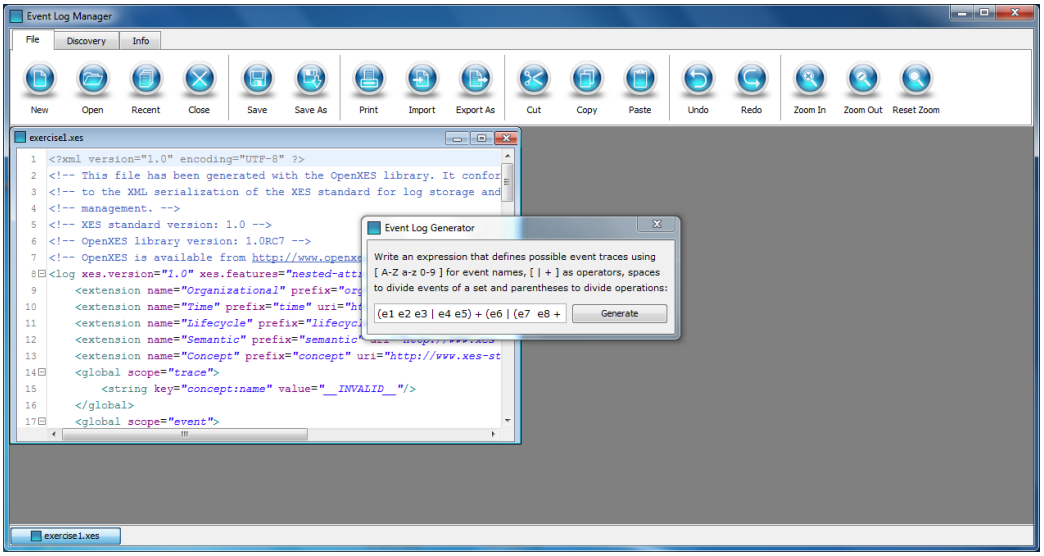

Figura 7.5: La finestra per l'inserimento del linguaggio di espressioni

Dopo aver premuto il pulsante "Generate" del pannello, il sistema mostrerà l'elenco delle tracce generate a partire dall'espressione passata, come si vede in Figura 7.6.

A questo punto l'utente avrà tre scelte a disposizione: scartare le tracce generate e modificare l'espressione per generarne di nuove, trasformare l'elenco di tracce in un documento CSV secondo la semantica definita precedentemente, oppure trasformare tale elenco in un log XES in formato XML.

In Figura 7.7 vengono mostrati tutti gli elementi di questo procedimento di generazione per espressioni: il pannello di inserimento dell'espressione, il log XES e il log in formato CSV generati da quest'ultima.

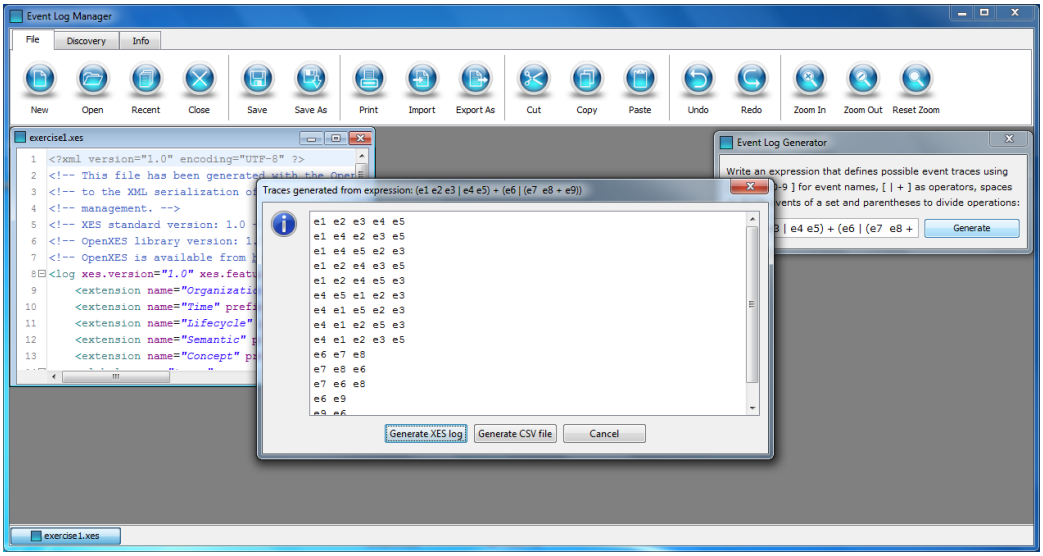

Figura 7.6: Il risultato della generazione tramite espressioni

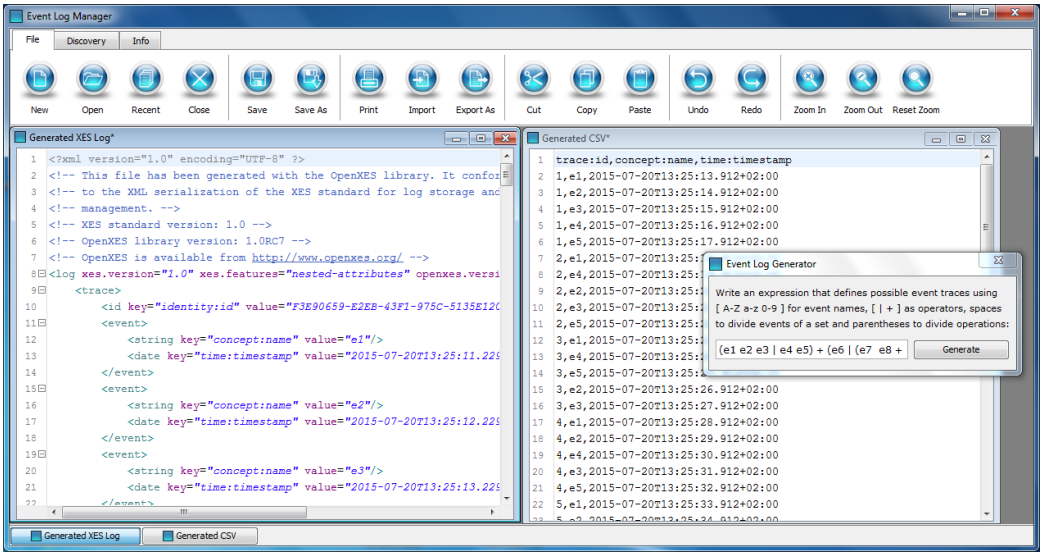

Figura 7.7: Il risultato della generazione utilizzando le espressioni

Per quanto riguarda la versione in formato XML, questa costituisce già la versione finale della generazione.

Per quanto riguarda invece la versione in formato CSV, questa è interme-

dia nel processo di generazione e serve per poter aggiungere ulteriori attributi agli eventi delle tracce, prima di trasformare il documento finale modificato secondo le esigenze in un log in formato XML, utilizzando la funzione "Generate" del pannello Discovery vista precedentemente.

Anche in questo caso l'utente può usufruire della funzione di completamento automatico per aggiungere altri campi agli eventi, richiamando tale funzione attraverso la scorciatoia CTRL + SPAZIO.

7.2 L'analisi di un log

Dopo aver generato un log di eventi, quello che l'utente può fare è validarlo, usando il pulsante "Validate" del menù Discovery.

Questa procedura validerà il documento e riporterà un messaggio positivo, nel caso il documento risulti valido rispetto allo Schema dello standard XES, un messaggio negativo nel caso contrario in cui abbia riscontrato degli errori (mostrando di che errori si tratta).

Nella Figura 7.8 è mostrato il caso più interessante nel quale la procedura di validazione ha esito negativo.

Dopo una validazione positiva, il log è pronto per essere analizzato.

L'utente può analizzare un log attraverso il pulsante "Footprint" del menù Discovery.

L'algoritmo di analisi procederà a controllare il log e restituirà alla fine dell'analisi un report come quello mostrato in Figura 7.9.

Per migliorare l'analisi iniziale del log, l'utente può a questo punto impostare dei filtri, in modo da visualizzare diverse prospettive di analisi.

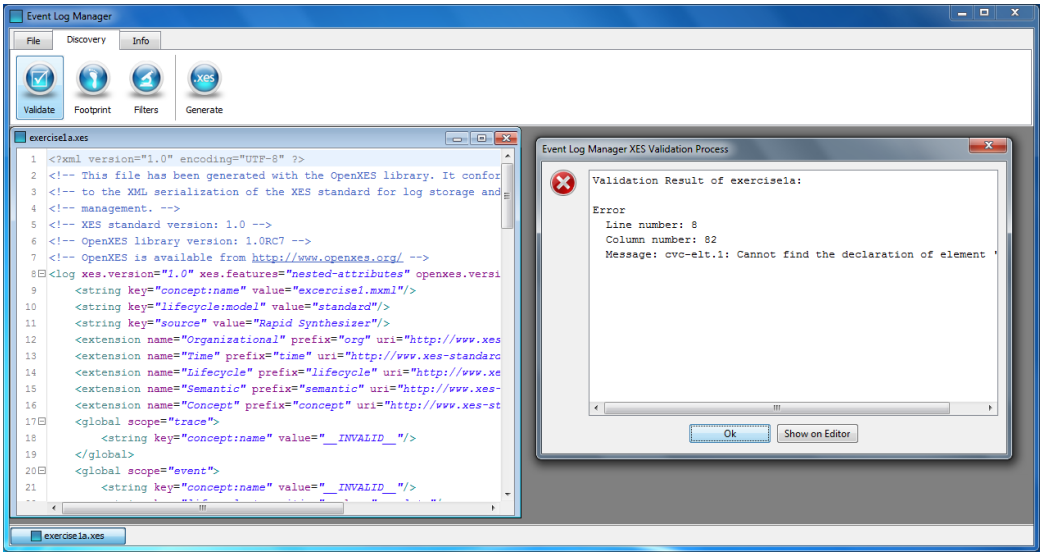

Figura 7.8: Un possibile risultato negativo della validazione

In Figura 7.10 viene mostrato il pannello che si apre premendo il pulsante "Filters" del menù Discovery.

Attraverso questo pannello l'utente può impostare i seguenti valori:

1. Il numero di occorrenze minino affinché una traccia possa essere considerata dall'analisi, cercando quindi di selezionare in questo modo solo un sottoinsieme di tracce che appaiono più spesso e che quindi hanno maggiore valore per l'analisi, piuttosto che considerare anche relazioni derivanti da tracce che ad esempio appaiono una sola volta.

2. Il classificatore di eventi da utilizzare, ovvero quale tra i classificatori definiti nel codice del log debba essere utilizzato per classificare gli eventi delle tracce in gruppi con gli stessi valori per gli attributi definiti dal classificatore; queste classi saranno considerate poi come gli oggetti

Footprint Matrix of exercise3

$\mathbf{L} = [\langle b, f, d, g \rangle^1, \langle a, c, e, g \rangle^1, \langle a, e, c, g \rangle^1, \langle b, d, f, g \rangle^1]$

	a	b	c	d	e	f	g
a	#	#	→	#	→	#	#
b	#	#	#	→	#	→	#
c	←	#	#	#	||	#	→
d	#	←	#	#	#	||	→
e	←	#	||	#	#	#	→
f	#	←	#	||	#	#	→
g	#	#	←	←	←	←	#

Filters

Selected Classifier: Event Name
Min Trace Occurences: 1

Matrix Legend

alias → concept:name
a → A
b → B
c → C
d → D
e → E
f → F
g → G

Figura 7.9: Un report di analisi

veri e propri di analisi, in quanto sarà di queste classi che la footprint matrix mostrerà le dipendenze causali.

3. Quali tracce considerare per l'analisi, in base ai loro eventi iniziali; le tracce che verranno considerate saranno solo quelle che inizieranno con un evento che risulta selezionato in questo pannello, altrimenti verranno

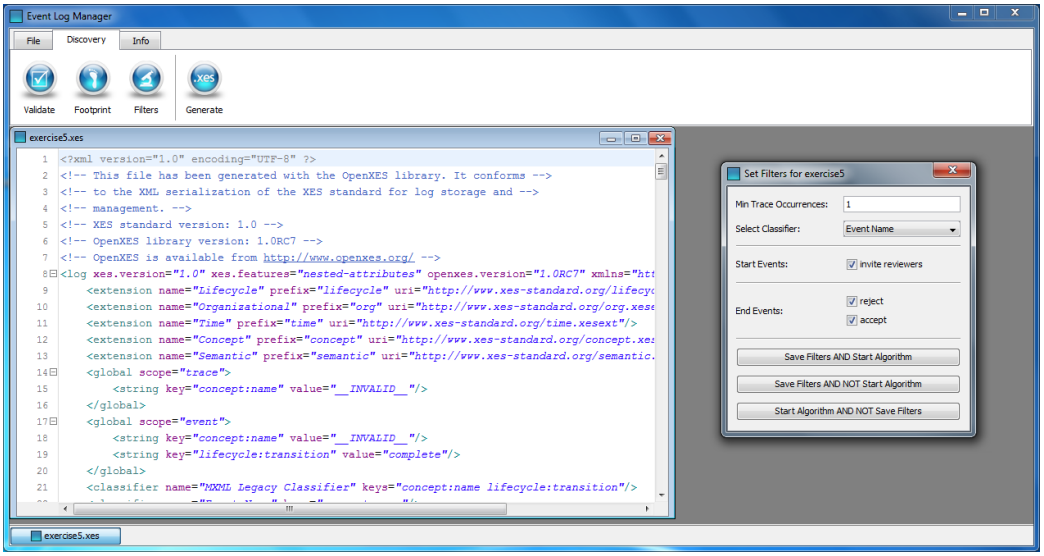

Figura 7.10: I filtri di analisi

scartate.

4. Quali tracce considerare per l'analisi, in base ai loro eventi finali, analogamente al ragionamento del punto precedente.

Dopo aver impostato un filtro, l'utente può scegliere tra tre soluzioni:

1. Salvare il filtro come di default per il log all'interno dell'editor selezionato ed avviare di nuovo l'analisi, producendo un nuovo report questa volta con i filtri impostati; il filtro salvato permetterà adesso di ottenere questo report di analisi anche dal pulsante "Footprint" del menù, senza dover passare per il pannello dei filtri impostando ogni volta i parametri desiderati.

2. Salvare il filtro ma non eseguire l'analisi; utile semplicemente per impostare un nuovo filtro come quello di default.

3. Non salvare il filtro come di default, ma avviare l'analisi utilizzando un filtro temporaneo definito attraverso i parametri impostati per ottenere il relativo report; questa funzione è molto utile nel caso si debbano fare delle prove con filtri diversi, senza necessariamente modificare quello di default.

Questo pannello è direttamente collegato ad uno specifico log, in quanto ne mostra il suo filtro.

Di conseguenza qualsiasi azione eseguita sull'editor (chiusura, minimizzazione, massimizzazione, selezione) comporta l'applicazione della stessa azione anche sul pannello.

Capitolo 8

CONCLUSIONI

L'obiettivo di questa tesi era la progettazione e lo sviluppo di una applicazione basata sullo standard XES per la creazione, modifica ed analisi di log di eventi memorizzati utilizzando questo formato.

Come abbiamo visto, l'applicazione dà la possibilità di aprire e modificare log di eventi reali utilizzando un editor molto efficace.

L'applicazione permette inoltre di generare nuovi log, utilizzando ad esempio le funzioni di Template e Completion dell'editor, oppure specificando, attraverso il linguaggio di espressioni definito all'interno del sistema, possibili sequenze di attività appartenenti ad una precisa esecuzione del processo.

La generazione di nuovi log può inoltre essere fatta anche attraverso l'utilizzo della semantica in formato CSV per la definizione di sequenze di attività connesse ad una stessa istanza di processo.

Molti sono quindi i modi messi a disposizione per la didattica attraverso i quali un utente può generare un nuovo log e successivamente gestirlo e

analizzarlo.

Il sistema sviluppato permette a questo punto la validazione e l'analisi di ogni log aperto e la generazione di un report di analisi completo e dettagliato, che ha il suo massimo significato nella visualizzazione della *footprint matrix* generata a partire dal log stesso: in questa matrice vengono mostrate le dipendenze causali tra le attività del log, dalle quali poi si possono trarre importanti conclusioni sull'esecuzione del processo analizzato.

Grazie ad un intuitivo e completo sistema di filtri, l'applicazione da modo all'utente di definire i parametri dell'analisi e ottenere un report ed una matrice di maggior interesse per l'analisi del processo.

L'obiettivo dell'analisi, come ricordato spesso in questo lavoro, è la ricerca della conoscenza sul processo in esame, a partire dalla registrazione delle sue esecuzioni, il log di eventi appunto.

Una volta elaborati i log e generati i report di analisi, l'applicazione dà la possibilità di esportare questi contenuti in formati diversi, oltre a quelli testuali che sono l'immediata rappresentazione del contenuto del documento all'interno di un file esterno.

Abbiamo visto infatti come un log possa essere esportato al di fuori del sistema utilizzando la semantica in formato CSV che abbiamo definito nel corso dello sviluppo.

Oppure, come un report di analisi possa essere esportato in formati diversi, a seconda delle esigenze e di quale sia l'utilizzo che intendiamo farne in seguito. I formati disponibili per questo contenuto sono: HTML, PDF, LaTeX e PNG.

Insieme a questa funzionalità di export, ne viene fornita una di import

nel sistema, per poter ricaricare documenti precedentemente esportati.

Durante la progettazione di questo sistema, numerose sono state le funzionalità che si è deciso di sviluppare (molte di queste non presenti in altre applicazioni di questo genere), al fine di rendere questo applicativo uno strumento utile per scopi didattici e interessante all'interno della disciplina del Process Mining e tra gli strumenti già sviluppati per l'analisi dei processi.

L'intento era infatti quello di fornire nuove funzionalità a questa area di ricerca e di integrarsi con le altre applicazioni già presenti in questo ambito.

Quello che è stato ottenuto al completamento del progetto è una applicazione facile da utilizzare, robusta, soprattutto per quanto riguarda la gestione di tutte le possibili situazioni che possono accadere durante il suo utilizzo, adatta alla gestione e all'analisi dei log sia in campo accademico sia in quello aziendale e professionale.

Per quanto riguarda la parte di progettazione e sviluppo inoltre, il progetto presenta codice organizzato in modo chiaro dal punto di vista architetturale, grazie all'utilizzo del pattern Modified Model-View-Controller, largamente commentato e documentato, con tutte le funzionalità implementate finemente interconnesse tra di loro.

Queste caratteristiche facilitano eventuali estensioni future dell'applicazione.

L'obiettivo a lungo termine che si propone questo progetto è quello di continuare a crescere in termini di funzionalità aggiunte e di utenza, con il chiaro intento di diventare un importante strumento all'interno della disciplina dell'analisi dei processi. Ad esempio, delle funzionalità che potrebbero essere facilmente incluse in futuro riguardano il confronto tra footprint matrix, per

vedere come piccoli cambiamenti sulle tracce si riflettono sulle dipendenze tra eventi e l'integrazione di altre tecniche di analisi dei log.

L'applicazione è stata sviluppata in Java e verrà rilasciata con licenza Open Source con il nome di *Event Log Manager*.

Bibliografia

[1] Wil M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[2] Wil M. P. van der Aalst, Arya Adriansyah, Ana Karla Alves de Medeiros, Franco Arcieri, Thomas Baier, et al. Process mining manifesto. In Florian Daniel, Kamel Barkaoui, and Schahram Dustdar, editors, *Business Process Management Workshops - BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I*, volume 99 of *Lecture Notes in Business Information Processing*, pages 169–194. Springer, 2011.

[3] Wil M. P. van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Trans. Knowl. Data Eng.*, 16(9):1128–1142, 2004.

[4] Apple. Concepts in objective-c programming - model-view-controller, January 2012. https://developer.apple.com/library/mac/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html.

[5] Robert Eckstein. Java se application design with mvc, March 2007. http://www.oracle.com/technetwork/articles/javase/index-142890.html.

[6] Christian W. Günther and Eric Verbeek. Openxes developer guide v 2.0, March 2014. http://www.xes-standard.org/_media/openxes/openxesdeveloperguide-2.0.pdf.

[7] Christian W. Günther and Eric Verbeek. Xes standard definition v. 2.0, March 2014. http://www.xes-standard.org/_media/xes/xesstandarddefinition-2.0.pdf.